C++ GUI Qt 4 编程

（第二版）

C++ GUI Programming with Qt 4

Second Edition

[加拿大] Jasmin Blanchette
[英] Mark Summerfield 著

闫锋欣 曾泉人 张志强 译

周莉娜 赵延兵 审校

電子工業出版社
Publishing House of Electronics Industry
北京 · BEIJING

内容简介

本书详细讲述了用最新的 Qt 版本进行图形用户界面应用程序开发的各个方面。前 5 章主要涉及 Qt 基础知识，后两个部分主要讲解 Qt 的中高级编程，包括布局管理、事件处理、二维/三维图形、拖放、项视图类、容器类、输入/输出、数据库、多线程、网络、XML、国际化、嵌入式编程等内容。对于本书讲授的大量 Qt 4 编程原理和实践，都可以轻易将其应用于 Qt 4.4、Qt 4.5 以及后续版本的 Qt 程序开发过程中。

本书适合对 Qt 编程感兴趣的程序员以及广大计算机编程爱好者阅读，也可作为相关机构的培训教材。

Authorized translation from the English language edition, entitled C++ GUI PROGRAMMNG WITH QT4, second edition, 9780132354165 by BLANCHETTE, JASMIN; SUMMERFIELD, MARK, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2008 Trolltech ASA.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright © 2008.

本书简体中文版由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2008-2499

图书在版编目(CIP)数据

C++ GUI Qt 4 编程：第 2 版/(加)布兰切特(Blanchette, J.)，(英)萨默菲尔德(Summerfield, M.)著；闫锋欣等译. 北京：电子工业出版社，2008.8

书名原文：C++ GUI Programming with Qt 4, 2nd edition

ISBN 978-7-121-07038-9

I. C… II. ①布…②萨…③闫… III. 软件工具－程序设计 IV. TP311.56

中国版本图书馆 CIP 数据核字(2008)第 099616 号

责任编辑：许菊芳

印　　刷：北京市天竺颖华印刷厂

装　　订：三河市金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　邮编　100036

开　　本：787×1092　1/16　印张：32　字数：820 千字

印　　次：2008 年 8 月第 1 次印刷

定　　价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：(010)88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：(010)88258888。

译 者 序

感谢奇趣科技公司(Trolltech,www.trolltech.com)为我们提供了 Qt。无论是职业的程序开发人员,还是编程爱好者,都希望自己编写的应用程序可以流畅地运行于所有平台,而 Qt 在这一方面的出众表现令我们印象深刻:利用 Qt 提供的 C++ 应用程序开发框架,可以轻松实现"一次编写,随处编译"的跨平台解决方案,使我们的应用程序能完美运行于从 Windows 98 到 Vista,从 Mac OS X 到 Linux,从 Solaris、HP-UX 到其他基于 X11 的众多 UNIX 平台之上。与此同时,作为 Qt 组成部分之一的 Qt/Embedded Linux,也为嵌入式系统的开发人员搭建了一套完善的窗口系统和开发平台。

Qt 具有功能强大的在线帮助文档系统。利用它,只需轻点鼠标或者简单敲击几下键盘,就可以轻易制作出简单的"Hello World"欢迎对话框,或者甚至是更为复杂的电子制表软件系统。这一点,在众多的软件帮助文档系统中并不多见。然而,帮助文档系统毕竟是以为用户提供实用的类库参考为主要目的的,也就是说,它主要是为用户提供准确的"可以如何做"的信息。这对于喜欢举一反三的程序设计人员来说,显然远远不够,因为我们更喜欢知道"为什么要这么做"。

本书围绕如何使用 Qt 编写图形用户界面程序这一中心,并尽可能多地采用手写代码的方式,生动、全面而又深刻地阐明了 Qt 程序的设计理念。当然,在本书以及上一版本相继出版的几年中,Qt 已经由一个简单的图形工具包演变成长为具有事实标准意义的应用程序开发框架。如今,蓬勃发展的 KDE 桌面环境和诺基亚公司对奇趣科技公司的收购,进一步展示了 Qt 的无限发展潜力和令人期待的远景。

本书作为"C++ GUI Programming with Qt 4"的第二版,在充分讲述 Qt 4.x 新特性的同时,又较好地延续了第一版的风格。全书仍由四部分构成,第一部分按照循序渐进、由浅入深的原则,介绍了使用 Qt 编写图形用户界面应用程序时所需的基本概念,并对这些基础知识安排了对应的实践训练,使读者仅利用这一部分知识就足以写出实用的图形用户界面应用程序。第二部分进一步深入介绍了 Qt 中的一些重要知识,如事件处理、拖放操作、项视图、多线程等。第三部分提供了更为专业和高级的内容,如三维图形、创建插件和应用程序脚本等。如果您已经较好地掌握了 Qt 的基础知识,那么完全可以像使用一本实用参考书一样以任意的顺序阅读第二部分和第三部分中的章节。第四部分由数个附录构成,分别介绍了 Qt 的获取和安装、Qt 应用程序的构建、Qt Jambi 和 C++ 的基础知识,它们可以帮助您更充分地使用 Qt。

正如 Prentice Hall 开源软件开发系列丛书主编之一的 Arnold Robbins 所说的那样,"这的确是一本好书"。首先,本书是由奇趣科技公司推出的关于 Qt 的官方书籍,也是该公司新员工的培训教材。当然,这只能算作一个佐证而已。其次,作者在介绍 Qt 程序设计中的很多关键内容时,不仅详细阐述了各种基本概念和底层背景知识,而且还对编程实践中的技巧和理论知识做了充分说明,它们是开发其他程序时可供借鉴的宝贵经验,这也使得本书成为一部很有价值的软件技术书籍。

在本书的翻译过程中,首先要感谢齐亮所做的巨大贡献。当我们在 2006 年联系他并且希望与他合译本书的第一版时,他就慷慨地接受了。几经辗转,当我们终于得到授权可以翻译第二版(也就是本书)时,他秉持无私的开源精神,无偿奉献了《C++ GUI Qt 3 编程》一书的电子版源文件以及第一版的部分译稿草稿,从而大大提高了翻译进度。齐亮作为国内 Qt 技术的布道者、执著而无私的开源技术人员(他是 KDE 开源项目的主要成员之一),值得我们每一个人学习。

本书翻译工作的具体分工是：西北工业大学的闫锋欣翻译了本书的第 1～7、18、21、23 章以及附录 A、附录 D、前言和致谢等部分；曾泉人翻译了第 9～17 章；奇趣科技公司北京分公司的张志强工程师翻译了本书的第 8、19、20、22、24 章以及附录 B 和附录 C。我们还邀请了西安欧亚学院的周莉娜和赵延兵两位老师作为本书的外部审稿人，他们的细致和耐心，为我们的工作增色不少。此外，参与本书文字校对工作的还有：西北工业大学的陆达方和丁士鹏先生，西安建筑科技大学的赵国锋硕士。还要感谢西北工业大学的张延超博士，他完成了本书第一版的审校工作。

为了更充分地使用本书，您除了可以访问原书的站点 http://www.informit.com/title/0132354160 外，还可以访问本书的中文站点 http://www.qtcn.org/gpq4/。在这些地方，您不仅可以下载到与本书配套的示例程序，还可以与各位专业人士一起讨论 Qt 的相关技术问题。这里要特别说明的是，非常感谢 www.qtcn.org 的网站负责人白建平（XChinux）先生，正是他才让本书得以在最为专业的 Qt 技术网站上占有一席之地。

由于译者水平有限，加之时间仓促，译稿中难免有曲解或误解作者原意的地方，望读者谅解并批评指正。同时，如果您有什么好的建议，可以随时在 http://www.qtcn.org/gpq4/上以留言的方式告诉我们。

译者

2008 年 5 月

丛书编者序

亲爱的读者：

作为一名职业程序员，我每天都在使用 Qt，而且我对 Qt 的组织结构、设计及其为 C++ 程序员所带来的动力印象深刻。

自 Qt 作为跨平台的图形用户界面工具包诞生以来，它已经扩展到了包括便携式设备在内的几乎当今程序设计的所有领域，比如文件、进程、网络和数据库的存取访问等。由于 Qt 的广泛适应性及良好的可移植性，只要您编写过一次代码，那么在其他不同的操作系统平台中，只需重新将其编译一遍即可重复使用。特别是当客户要求您的应用程序能同时运行于不同的平台时，Qt 的这一优势就显得格外重要了。

当然，使用开源许可协议也可以获得 Qt。如果您是一名开源程序开发人员，那么，从 Qt 那里也将获益无穷。

尽管 Qt 提供了大量的在线帮助文档，但这些帮助文档以参考性内容为主。示例程序非常有用，但仅仅通过阅读这些示例程序就希望能够在自己的程序中正确使用 Qt，显然是一件非常困难的事情。而这一点正是本书引人入胜的地方所在。

这的确是一本好书。首先，这是一本由奇趣科技公司推出的关于 Qt 的官方书籍，这说起来可能有些多余了。同时，它也是一部巨著：组织巧妙、文笔优雅，并且易于根据该书进行学习。与阐述伟大技术的巨著相结合，更易于造就真正的成功者，而这也就是为什么让我感到非常自豪和非常兴奋并乐于将此书作为 Prentice Hall 开源软件开发系列丛书之一的原因。

我希望您能享受到阅读这本书并从中获益良多的那种乐趣，而我，的确已经体会到了这一点。

Arnold Robbins
Nof Ayalon，以色列
2007 年 11 月

前　言

为什么会是 Qt？为什么像我这样的程序员会选择 Qt？这个问题的答案显而易见：Qt 单一源程序的兼容性、丰富的特性、C++ 方面的性能、源代码的可用性、它的文档、高质量的技术支持，以及在奇趣科技公司那些精美的营销材料中所涉及的其他优势等。这些答案看起来确实都不错，但是遗漏了最为重要的一点：Qt 的成功缘于程序员们对它的喜欢。

那么，是什么让程序员喜欢某种技术而放弃另外一种呢？就我而言，我认为软件工程师们喜欢某种技术，是因为他们觉得这种技术是合适的，但是这也会让他们讨厌所有那些他们觉得不合适的其他技术。除此之外，我们还能解释下面的这些情况吗？例如，一些最出众的程序员需要在帮助之下才能编写出一个录像机程序，或者又比如，似乎大多数工程师在操作本公司的电话系统时总会遇到麻烦。我虽然善于记住随机数字和指令的序列，但是如果将其比作用于控制我的应答系统所需要的条件来说，则可能一条也不具备。在奇趣科技公司，我们的电话系统要求在拨打其他人的分机号码前，一定要按住“*”键 2 秒后才允许开始拨号。如果忘记了这样做而是直接拨打分机号码，那么就不得不再重新拨一遍全部的号码。为什么是“*”键而不是“#”键、“1”键或者“5”键？或者为什么不是 20 个电话键盘中的其他任何一个呢？又为什么是 2 秒，而不是 1 秒、3 秒或者 1.5 秒呢？问题到底出在哪里？我发现电话很气人，所以我尽可能不去使用它。没有人喜欢总是去做一些不得不做的随机事情，特别是当这些随机事情显然只出现在同样随机的情况下的时候，真希望自己从来都没有听到过它。

编程很像我们正在使用的电话系统，并且要比它还糟糕。而这正是 Qt 所要解决的问题。Qt 与众不同。一方面，Qt 很有意义；另一方面，Qt 颇具趣味性。Qt 可以让您把精力集中在您的任务上。当 Qt 的首席体系结构设计师面对一个问题的时候，他们不是寻求一个好的、快速的或者最简便的解决方案，而是在寻求一个恰当的解决方案，然后将其记录在案。应当承认，他们犯下了一些错误，并且还要承认的是，他们的一些设计决策没有通过时间的检验，但是他们确实做出了很多正确的设计，并且那些错误的设计应当而且也是能够进行改正的。看一看最初设计用于构建 Windows 95 和 UNIX Motif 之间的桥梁系统，到后来演变为跨越 Windows Vista、Mac OS X 和 GNU/Linux 以及那些诸如移动电话等小型设备在内的统一的现代桌面系统，这些事实就足以证明这一点。

早在 Qt 大受欢迎并且被广泛使用很久以前，正是 Qt 的开发人员为寻求恰当的解决方案所做出的贡献才使 Qt 变得与众不同。其贡献之大，至今仍然影响着每一个对 Qt 进行开发和维护的人。对我们而言，研发 Qt 是一种使命和殊荣。能够使您的职业生涯和开源生活变得更为轻松和更加有趣，这让我们倍感自豪。

人们乐于使用 Qt 的诸多原因之一是它的在线帮助文档，但是该帮助文档的主要目的是集中介绍个别的类，而很少讲述应当如何构建现实世界中那些复杂的应用程序。这本好书填补了这一缺憾，它展示了 Qt 所提供的东西，如何使用“Qt 的方式”进行 Qt 编程，以及如何充分地利用 Qt。本书将指导 C++、Java 或者 C# 程序员进行 Qt 编程，并且提供了丰富详实的资料来使他们成长为老练的 Qt 程序员。这本书包含了很多很好的例子、建议和说明——并且，该书也是我们对那些新加入公司的程序员们进行培训的入门教材。

如今，已有大量的商业或者免费的 Qt 应用程序可以购买或者下载，其中的一些专门用于特殊的高端市场，其他一些则面向大众市场。看到如此多的应用程序都是基于 Qt 构建而成的，这使我们充满了自豪感，并且还激励我们要让 Qt 变得更好。相信在这本书的帮助下，将会前所未有地出现更多的、质量更高的 Qt 应用程序。

Matthias Ettrich
德国，柏林
2007 年 11 月

序　言

Qt使用“一次编写，随处编译”的方式为开发跨平台的图形用户界面应用程序提供了一个完整的C++应用程序开发框架。Qt允许程序开发人员使用应用程序的单一源码树来构建可以运行在不同平台下的应用程序的不同版本；这些平台包括从Windows 98到Vista、Mac OS X、Linux、Solaris、HP-UX以及其他很多基于X11的UNIX。许多Qt库和工具也都是Qt/Embedded Linux的组成部分。Qt/Embedded Linux是一个可以在嵌入式Linux上提供窗口系统的产品。

本书的目标就是教您如何使用Qt 4来编写图形用户界面程序。本书从“Hello Qt”开始，然后很快地转移到更高级的话题中，如自定义窗口部件的创建和拖放功能的提供等。通过本书的互联网站点（http://www.informit.com/title/0132354160），您可以下载到一些作为本书文字补充材料的示例程序。附录A说明了如何下载和安装这些软件，其中包括一个用于Windows的C++免费编译器。

本书分为四部分。第一部分涵盖了在使用Qt编写图形用户界面应用程序时所必需的全部基本概念和练习。仅掌握这一部分中所蕴含的知识就足以写出实用的图形用户界面应用程序。第二部分进一步深入介绍了Qt的一些重要主题，第三部分则提供了更为专业和高级的材料。您可以按任意顺序阅读第二部分和第三部分中的章节，但这是建立在您对第一部分中的内容非常熟悉的基础之上的。第四部分包括数个附录，附录B说明了如何构建Qt应用程序，附录C则介绍了Qt Jambi，它是Java版的Qt。

本书的第一版建立在Qt 3版本的基础上，尽管已通过全书修订来反映那些很好的Qt 4编程技术，但本书还是根据Qt 4的模型/视图结构、新的插件框架、使用Qt/Embedded Linux进行嵌入式编程等内容而引入了一些新的章节和一个新的附录。作为第二版，本书充分利用了Qt 4.2和Qt 4.3中引入的新特性对其进行了彻底更新，并包含“自定义外观”和“应用程序脚本”两个新的章以及两个新的附录。原有的“图形”一章已经拆分为“二维”和“三维”两章，在它们中间，涵盖了新的图形视图类和QPainter的OpenGL后端实现。此外，在数据库、XML和嵌入式编程等几章中，还添加了许多新内容。

与本书的前两版一样，这一版的重点放在如何进行Qt编程的说明和许多真实例子的提供上，而不是对丰富的Qt在线文档的简单拼凑和总结。因为本书纯粹讲授的是Qt 4编程中的原理和实践知识，因而读者能够轻松学会将要出现在Qt 4.4、Qt 4.5以及Qt 4.x等后续版本中的15个Qt新模块。如果您正在使用的Qt版本恰好是这些后续版本中的一个，那么当然要阅读一下参考文档中的“What's New in Qt 4.x”一章，以便可以对那些可用的新特性有一个总体把握。

在写作本书的时候，是假定您已经具备了C++、Java或者C#的基本知识。本书中的例子代码使用的是C++中的一个子集，从而避免了很多在Qt编程中极少使用的C++特性。在某些不可避免而必须使用C++高级结构的地方，会在使用时对其做出必要的解释。如果您对Java或者C#已经非常熟悉但是对C++还知之不多甚至一无所知，那么建议您先阅读附录D。附录D提供了对C++较为充分的介绍，从而能够让您具有使用本书所必备的C++知识。对于C++中的面向对象编程更为全面的介绍，建议您阅读由P. J. Deitel和H. M. Deitel编著的“C++ How to Program”（Prentice Hall，2007），以及由Stanley B. Lippman、Josée Lajoie和Barbara E. Moo编著的“C++ Primer”（Addison-Wesley，2005）这两本书。

Qt 以其作为一个跨平台框架而著称，但由于 Qt 拥有直观、强大的 API(应用程序编程接口)，很多公司更愿意把 Qt 用于单一平台的软件开发上。Adobe PhotoShop Album 就是用 Qt 编写的面向大众市场的 Windows 应用程序中的一个例子。纵向市场中很多功能完善的软件系统，如三维动画工具、数字电影处理软件、自动化电路设计系统(用于芯片设计)、油气资源勘探、金融服务以及医学成像等，都可以基于 Qt 构建而成。如果您正是一名通过 Qt 成功编写 Windows 产品来谋求发展的人，那么只需通过重新编译您的产品，就可以轻松地在 Mac OS X 和 Linux 世界中开拓出新的市场。

可以基于多种许可协议获得 Qt 的使用权。如果想构建商业应用程序，那么必须从奇趣科技公司购买一个 Qt 的商业许可协议。但如果只想构建一些开源程序，那么就可以使用基于 GPL 的 Qt 开源版本。KDE 和多数开源应用程序都是基于这种模式并使用 Qt 构建起来的。

除了 Qt 的数百个类之外，还有很多扩展 Qt 应用范围和功能的其他软件。其中的一些产品，像 Qt Solutions 中的一些组件，都可以从该公司获得。同时，还有其他很多软件由另外一些公司或者开源社区提供。对于可用的 Qt 额外软件的列表清单，可以查阅 http://www.trolltech.com/products/qt/3rdparty/。奇趣科技公司的开发人员也有他们自己的网站，这就是 Trolltech 实验室(Trolltech Labs，http://labs.trolltech.com/)，他们会把自己写的一些用于娱乐方面的、有趣的或者是有用的非官方代码放在那里。Qt 还建立了一个维护良好并且内容丰富的用户社区，用户可以通过 Qt 兴趣邮件列表来进行交流，详细情况请参阅 http://lists.trolltech.com/。

如果您在本书中发现了任何错误、对下一版有任何建议或者想反馈意见，我们将非常高兴收到您的邮件，邮件请发送到 qt-book@trolltech.com。

致　谢

首先要感谢 Eirik Chambe-Eng，他是奇趣科技公司的总裁，也是该公司两位创始人之一。Eirik 不仅热情地鼓励我们编写了本书的 Qt 3 版本一书，他还允许我们在写作本书时占用大量的工作时间。Eirik 和公司的 CEO——Haavard Nord，都阅读了本书的初稿并给出了许多宝贵的意见。他们的慷慨和远见来自于 Matthias Ettrich，他是该公司的首席程序员，他欣然同意我们把编著本书放在工作的首位，并且给我们提出了很多关于 Qt 编程良好风格习惯的建议。

在本书的 Qt 3 版中，我们邀请了 Qt 的两位客户：Paul Curtis 和 Klaus Schmidinger，由他们作为我们的外部审稿人。他们都是 Qt 专家，非常关注技术细节，他们在初稿中发现了一些小的错误，并且提出了许多改进的建议。此外，在公司中，除了 Matthias 以外，Reginald Stadlbauer 也是我们最忠实的审稿人，他的技术洞察力是无价的，并且他还教会了我们如何在 Qt 中做一些我们甚至认为是不可能的事情。

对基于 Qt 4 的本书，我们继续从 Eirik、Haavard 和 Matthias 他们那里得到了无私的帮助和大力的支持。Klaus Schmidinger 继续给出了颇有价值的反馈意见，并且我们也继续从 Qt 的客户 Paul Floyd 对于一些新材料内容的详细审阅中获益。还要感谢 David García Garzón 在附录 B 中关于 SCons 的帮助。在公司里，其他的主要审稿人还有 Carlos Manuel Duclos Vergara、Andreas Aardal Hanssen、Henrik Hartz、Martin Jones、Vivi Glückstad Karlsen、Trond Kjernåsen、Trenton Schulz、Andy Shaw、Gunnar Sletta 和 Pål de Vibe。

除了以上所提到的几位审稿人之外，我们还得到了一些专家的帮助，他们分别是：Eskil Abrahamsen Blomfeldt（Qt Jambi）、Frans Englich（XML）、Harald Fernengel（数据库）、Kent Hansen（应用程序脚本）、Volker Hilsheimer（ActiveX）、Bradley Hughes（多线程）、Lars Knoll（二维图形和国际化）、Anders Larsen（数据库）、Sam Magnuson（qmake）、Marius Bugge Monsen（项视图类）、Dimitri Papadopoulos（Qt/X11）、Girish Ramakrishnan（样式表）、Samuel Rødal（三维图形）、Rainer Schmid（网络和 XML）、Amrit Pal Singh（C++ 简介）、Paul Olav Tvete（自定义窗口部件和嵌入式编程）、Geir Vattekar（Qt Jambi）和 Thomas Zander（编译系统）。

写作本书占用了我们大量的时间。同时，还要对公司的文档以及与处理文档相关的技术支持团队表示感谢，并且也要对公司的系统管理员们表示感谢，他们让我们的机器始终稳定运行并可以随时与整个项目的网络保持联系。

在本书的制作方面，感谢 Lout 排版系统工具的作者 Jeff Kingston，他连续不断地增强了该工具的许多功能，并且对我们的许多问题给予了及时的回复。也要感谢 James Cloos 所提供的 DejaVu Mono 字体的压缩版，该字体是我们使用的 monospaced 字体的基础。感谢奇趣科技公司的 Cathrine Bore 代表我们处理了合同和法律上的事务。还要感谢 Nathan Clement 所做的系列插图，以及 Audrey Doyle 对本书所做的详细校对。最后，感谢我们的编辑 Debra Williams-Cauley，既要感谢她的支持，又要感谢她为我们提供的轻松自在的写作进程。还要感谢 Lara Wysong 编辑，它使得本书这么实用。

Qt 简史

Qt 框架首度为公众可用是在 1995 年 5 月。它最初由 Haavard Nord(奇趣科技公司的 CEO)和 Eirik Chambe-Eng(公司总裁)开发而成。Haavard 和 Eirik 在位于挪威特隆赫姆的挪威科技学院相识,在那里,他们都获得了计算机科学的硕士学位。

Haavard 对 C++ 图形用户界面开发的兴趣始于 1988 年,当时一家瑞典公司委托他开发一套 C++ 图形用户界面框架。几年后,在 1990 年的夏天,Haavard 和 Eirik 因为一个超声波图像方面的 C++ 数据库应用程序而在一起工作。这个系统需要一个能够在 UNIX、Macintosh 和 Windows 上都能运行的图形用户界面。在那个夏天中的某天,Haavard 和 Eirik 一起出去散步,享受阳光,当他们坐在公园的一条长椅上时,Haavard 说:"我们需要一个面向对象的显示系统。"由此引发的讨论,为他们即将创建的面向对象的、跨平台的图形用户界面框架奠定了智力基础。

1991 年,Haavard 和 Eirik 开始一起合作设计、编写最终成为 Qt 的那些类。在随后的一年中,Eirik 提出了"信号和槽"的设想——一个简单并且有效的强大的图形用户界面编程规范,而现在,它已经可以被多个工具包实现。Haavard 实践了这一想法,并且据此创建了一个手写代码的实现系统。到 1993 年,Haavard 和 Eirik 已经开发出了 Qt 的第一套图形内核程序,并且能够利用它实现他们自己的一些窗口部件。同年末,为了创建"世界上最好的 C++ 图形用户界面框架",Haavard 提议一起进军商业领域。

1994 年成为两位年轻程序员不幸的一年,他们没有客户,没有资金,只有一个未完成的产品,但是他们希望能够闯进一个稳定的市场。幸运的是,他们的妻子都有工作并且愿意为他们的丈夫提供支持。在这两年里,Haavard 和 Eirik 认为,他们需要继续开发产品并且从中赚得收益。

之所以选择字母"Q"作为类的前缀,是因为该字母在 Haavard 的 Emacs 字体中看起来非常漂亮。随后添加的字母"t"代表"工具包"(toolkit),这是从"Xt"——一个 X 工具包的命名方式中获得的灵感。公司于 1994 年 3 月 4 日成立,最初的名字是"Quasar Technologies",随后更名为"Troll Tech",而公司今天的名字则是"Trolltech"。

1995 年 4 月,通过 Haavard 就读过的大学的一位教授的联系,挪威的 Metis 公司与他们签订了一份基于 Qt 进行软件开发的合同。大约在同一时间,公司雇佣了 Arnt Gulbrandsen,在公司工作的 6 年时间里,他设计并实现了一套独具特色的文档系统,并且对 Qt 的代码也做出了不少贡献。

1995 年 5 月 20 日,Qt 0.90 被上传到 sunsite.unc.edu。6 天后,在 comp.os.linux.announce 上发布。这是 Qt 的第一个公开发行版本。Qt 既可以用于 Windows 上的程序开发,又可以用于 UNIX 上的程序开发,而且在这两种平台上,都提供了相同的应用程序编程接口。从第一天起,Qt 就提供了两个版本的软件许可协议:一个是进行商业开发所需的商业许可协议版,另一个则是适用于开源开发的自由软件许可协议版。Metis 的合同确保了公司的发展,然而,在随后长达 10 个月的时间内,再没有任何人购买 Qt 的商业许可协议。

1996 年 3 月,欧洲航天局(European Space Agency)购买了 10 份 Qt 的商业许可协议,它成了第二位 Qt 客户。凭着坚定的信念,Eirik 和 Haavard 又雇佣了另外一名开发人员。Qt 0.97 在同年5 月底正式发布,随后在 1996 年 9 月 24 日,Qt 1.0 正式面世。到了这一年的年底,Qt 的版本已经发展到了 1.1,共有来自 8 个不同国家的客户购买了 18 份 Qt 的商业许可协议。也就是在这一年,在 Matthias Ettrich 的带领下,创立了 KDE 项目。

Qt 1.2 于 1997 年 4 月发布。Matthias Ettrich 利用 Qt 建立 KDE 的决定，使 Qt 成为 Linux 环境下开发 C++ 图形用户界面的事实标准。Qt 1.3 于 1997 年 9 月发布。

Matthias 在 1998 年加入公司，并且在当年 9 月，发布了 Qt 1 系列的最后一个版本——V 1.40。1999 年 6 月，Qt 2.0 发布，该版本拥有一个新的开源许可协议——Q 公共许可协议（QPL，Q Public License），它与开源的定义一致。1999 年 8 月，Qt 赢得了 LinuxWorld 的最佳库/工具奖。大约在这个时候，Trolltech Pty Ltd（澳大利亚）成立了。

2000 年，公司发布了 Qt/Embedded Linux，它用于 Linux 嵌入式设备。Qt/Embedded Linux 提供了自己的窗口系统，并且可以作为 X11 的轻量级替代产品。现在，Qt/X11 和 Qt/Embedded Linux 除了提供商业许可协议之外，还提供了广为使用的 GNU 通用公共许可协议（GPL，General Public License）。2000 年底，成立了 Trolltech Inc.（美国），并发布了 Qtopia 的第一版，它是一个用于移动电话和掌上电脑（PDA）的环境平台。Qt/Embedded Linux 在 2001 年和 2002 年两次获得了 LinuxWorld 的“Best Embedded Linux Solution”奖，Qtopia Phone 也在 2004 年获得了同样的荣誉。

2001 年，Qt 3.0 发布。现在，Qt 已经可用于 Windows、Mac OS X、UNIX 和 Linux（桌面和嵌入式）平台。Qt 3 提供了 42 个新类和超过 500 000 行的代码。Qt 3 是自 Qt 2 以来前进历程中最为重要的一步，它主要在诸多方面进行了众多改良，包括本地化和统一字符编码标准的支持、全新的文本查看和编辑窗口部件，以及一个类似于 Perl 正则表达式的类等。2002 年，Qt 3 赢得了 Software Development Times 的“Jolt Productivity Award”①。

2005 年夏，Qt 4.0 发布，它大约有 500 个类和 9000 多个函数，Qt 4 比以往的任何一个版本都要全面和丰富，并且它已经裂变成多个函数库，从而使开发人员可以根据自己的需要只连接所需要的 Qt 部分。相对于以前的所有 Qt 版本，Qt 4 的进步是巨大的，它不仅彻底地对高效易用的模板容器、高级的模型/视图功能、快速而灵活的二维绘图框架和强大的统一字符编码标准的文本查看和编辑类进行了大量改进，就更不必说对那些贯穿整个 Qt 类中的成千上万个小的改良了。现如今，Qt 4 具有如此广泛的特性，以至于 Qt 已经超越了作为图形用户界面工具包的界限，逐渐成长为一个成熟的应用程序开发框架。Qt 4 也是第一个能够在其所有可支持的平台上既可用于商业开发又可用于开源开发的 Qt 版本。

同样在 2005 年，公司在北京开设了一家办事处，以便为中国及其销售区域内的用户提供服务和培训，并且为 Qt/Embedded Linux 和 Qtopia 提供技术支持。

通过获取一些非官方的语言绑定件（language bindings），非 C++ 程序员也已早就开始使用 Qt，特别是用于 Python 程序员的 PyQt 语言绑定件。2007 年，公司发布了用于 C# 程序员的非官方语言绑定件 Qyoto。同一年，Qt Jambi 投放市场，它是一个官方支持的 Java 版 Qt 应用程序编程接口。附录 C 提供了对 Qt Jambi 的介绍。

自奇趣科技公司诞生以来，Qt 的声望经久不衰，而且至今依旧持续高涨。取得这样的成绩不但说明了 Qt 的质量，而且也说明了人们都喜欢使用它。在过去的 10 年中，Qt 已经从一个只被少数专业人士所熟悉的“秘密”产品，发展了到如今遍及全世界拥有数以千计的客户和数以万计的开源开发人员的产品。

① Jolt 大奖素有“软件业界的奥斯卡”之美誉，共设通用类图书、技术类图书、语言和开发环境、框架库和组件、开发者网站等十余个分类，每个分类设有一个“震撼奖”（Jolt Award）和三个“生产力奖”（Productivity Award）。一项技术产品只有在获得了 Jolt 奖之后才能真正成为行业的主流，一本技术书籍只有在获得了 Jolt 奖之后才能真正奠定其作为经典的地位。虽然 Jolt 奖项并不起决定作用，但它代表了某种技术趋势与潮流——译者注。

目　录

第一部分　Qt 基础

第 1 章　Qt 入门 …… 2
1.1　Hello Qt …… 2
1.2　建立连接 …… 4
1.3　窗口部件的布局 …… 4
1.4　使用参考文档 …… 7

第 2 章　创建对话框 …… 10
2.1　子类化 QDialog …… 10
2.2　深入介绍信号和槽 …… 15
2.3　快速设计对话框 …… 17
2.4　改变形状的对话框 …… 23
2.5　动态对话框 …… 29
2.6　内置的窗口部件类和对话框类 …… 30

第 3 章　创建主窗口 …… 35
3.1　子类化 QMainWindow …… 35
3.2　创建菜单和工具栏 …… 39
3.3　设置状态栏 …… 43
3.4　实现 File 菜单 …… 44
3.5　使用对话框 …… 49
3.6　存储设置 …… 54
3.7　多文档 …… 55
3.8　程序启动画面 …… 57

第 4 章　实现应用程序的功能 …… 59
4.1　中央窗口部件 …… 59
4.2　子类化 QTableWidget …… 60
4.3　载入和保存 …… 64
4.4　实现 Edit 菜单 …… 66
4.5　实现其他菜单 …… 70
4.6　子类化 QTableWidgetItem …… 73

第 5 章　创建自定义窗口部件 …… 80
5.1　自定义 Qt 窗口部件 …… 80
5.2　子类化 QWidget …… 81

5.3 在 Qt 设计师中集成自定义窗口部件 …… 89
5.4 双缓冲 …… 92

第二部分 Qt 中级

第 6 章 布局管理 …… 108
6.1 在窗体中摆放窗口部件 …… 108
6.2 分组布局 …… 113
6.3 切分窗口 …… 114
6.4 滚动区域 …… 117
6.5 停靠窗口和工具栏 …… 118
6.6 多文档界面 …… 121

第 7 章 事件处理 …… 129
7.1 重新实现事件处理器 …… 129
7.2 安装事件过滤器 …… 133
7.3 处理密集时的响应保持 …… 135

第 8 章 二维图形 …… 138
8.1 用 QPainter 绘图 …… 138
8.2 坐标系统变换 …… 142
8.3 用 QImage 高质量绘图 …… 148
8.4 基于项的图形视图 …… 150
8.5 打印 …… 166

第 9 章 拖放 …… 173
9.1 使拖放生效 …… 173
9.2 支持自定义拖动类型 …… 177
9.3 剪贴板处理技术 …… 181

第 10 章 项视图类 …… 182
10.1 使用项视图的简便类 …… 183
10.2 使用预定义模型 …… 187
10.3 实现自定义模型 …… 191
10.4 实现自定义委托 …… 203

第 11 章 容器类 …… 207
11.1 连续容器 …… 207
11.2 关联容器 …… 213
11.3 通用算法 …… 216
11.4 字符串、字节数组和变量 …… 217

第 12 章 输入与输出 …… 223
12.1 读取和写入二进制数据 …… 223
12.2 读取和写入文本 …… 228
12.3 遍历目录 …… 232

12.4 嵌入资源 …… 233
12.5 进程间通信 …… 234
第 13 章 数据库 …… 238
13.1 连接和查询 …… 238
13.2 查看表 …… 243
13.3 使用窗体编辑记录 …… 245
13.4 在表中显示数据 …… 249
第 14 章 多线程 …… 255
14.1 创建线程 …… 255
14.2 同步线程 …… 258
14.3 与主线程通信 …… 263
14.4 在次线程中使用 Qt 的类 …… 268
第 15 章 网络 …… 270
15.1 写 FTP 客户端 …… 270
15.2 写 HTTP 客户端 …… 277
15.3 写 TCP 客户/服务器应用程序 …… 279
15.4 发送和接收 UDP 数据报 …… 287
第 16 章 XML …… 291
16.1 使用 QXmlStreamReader 读取 XML …… 291
16.2 用 DOM 读取 XML …… 297
16.3 使用 SAX 读取 XML …… 300
16.4 写入 XML …… 304
第 17 章 提供在线帮助 …… 306
17.1 工具提示、状态提示和“What's This?”帮助 …… 306
17.2 利用 Web 浏览器提供在线帮助 …… 307
17.3 将 QTextBrowser 作为简单的帮助引擎 …… 309
17.4 使用 Qt Assistant 提供强大的在线帮助 …… 311

第三部分 Qt 高级

第 18 章 国际化 …… 314
18.1 使用 Unicode …… 314
18.2 让应用程序感知翻译 …… 317
18.3 动态切换语言 …… 322
18.4 翻译应用程序 …… 326
第 19 章 自定义外观 …… 329
19.1 使用 Qt 样式表 …… 329
19.2 子类化 QStyle …… 340
第 20 章 三维图形 …… 352
20.1 使用 OpenGL 绘图 …… 352

20.2 OpenGL 和 QPainter 的结合 …… 356
20.3 使用帧缓存对象生成叠加 …… 361

第 21 章 创建插件 …… 366
21.1 利用插件扩展 Qt …… 366
21.2 使应用程序感知插件 …… 374
21.3 编写应用程序的插件 …… 377

第 22 章 应用程序脚本 …… 379
22.1 ECMAScript 语言概述 …… 379
22.2 使用脚本扩展 Qt 应用程序 …… 386
22.3 使用脚本实现 GUI 扩展 …… 389
22.4 使用脚本自动化处理任务 …… 394

第 23 章 平台相关特性 …… 403
23.1 连接本地的应用程序编程接口 …… 403
23.2 在 Windows 上使用 ActiveX …… 406
23.3 处理 X11 会话管理 …… 416

第 24 章 嵌入式编程 …… 421
24.1 从 Qt/Embedded Linux 开始 …… 421
24.2 自定义 Qt/Embedded Linux …… 423
24.3 Qt 应用程序与 Qtopia 的集成 …… 424
24.4 使用 Qtopia 的 API …… 427

第四部分 附录

附录 A Qt 的获取和安装 …… 436
A.1 协议说明 …… 436
A.2 Qt/Windows 的安装 …… 436
A.3 Qt/Mac 的安装 …… 437
A.4 Qt/X11 的安装 …… 437

附录 B 编译 Qt 应用程序 …… 439
B.1 使用 qmake …… 439
B.2 使用第三方编译工具 …… 443

附录 C Qt Jambi 简介 …… 447
C.1 Qt Jambi 入门 …… 447
C.2 在 Eclipse IDE 中使用 Qt Jambi …… 451
C.3 在 Qt Jambi 中集成 C++ 组件 …… 455

附录 D 面向 Java 和C# 程序员的C++ 简介 …… 461
D.1 C++ 入门 …… 461
D.2 主要语言之间的差异 …… 465
D.3 C++ 标准库 …… 489

清华大学图书馆
C22591397



第一部分 Qt 基础

第 1 章 Qt 入门
第 2 章 创建对话框
第 3 章 创建主窗口
第 4 章 实现应用程序的功能
第 5 章 创建自定义窗口部件

第1章 Qt入门

这一章介绍了如何把基本的C++知识与Qt所提供的功能组合起来创建一些简单的图形用户界面(Graphical User Interface,GUI)应用程序。在这一章中,还引入了Qt中的两个重要概念:一个是"信号和槽",另外一个是"布局"。第2章还将对它们做进一步的阐述,而第3章将着手创建一个具有真正意义的应用程序。

如果你已经熟知Java或C#,但对C++的编程经验还有些欠缺的话,那么在开始阅读本书之前,可能需要先阅读附录D,它对C++做了简要介绍。

1.1 Hello Qt

我们先从一个非常简单的Qt程序开始。首先一行一行地研究这个程序,然后将会看到如何编译并运行它。

```
1 #include <QApplication>
2 #include <QLabel>

3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     QLabel *label = new QLabel("Hello Qt!");
7     label->show();
8     return app.exec();
9 }
```

第1行和第2行包含了类QApplication和QLabel的定义。对于每个Qt类,都有一个与该类同名(且大写)的头文件,在这个头文件中包括了对该类的定义。

第5行创建了一个QApplication对象,用来管理整个应用程序所用到的资源。这个QApplication构造函数需要两个参数,分别是argc和argv,因为Qt支持它自己的一些命令行参数。

第6行创建了一个显示"Hello Qt!"的QLabel窗口部件(widget)。在Qt和UNIX的术语(terminology)中,窗口部件就是用户界面中的一个可视化元素。该词起源于"window gadget"(窗口配件)这两个词,它相当于Windows系统术语中的"控件"(control)和"容器"(container)。按钮、菜单、滚动条和框架都是窗口部件。窗口部件也可以包含其他窗口部件,例如,应用程序的窗口通常就是一个包含了一个QMenuBar、一些QToolBar、一个QStatusBar以及一些其他窗口部件的窗口部件。绝大多数应用程序都会使用一个QMainWindow或者一个QDialog来作为它的窗口,但Qt是如此灵活,以至于任意窗口部件都可以用作窗口。在本例中,就是用窗口部件QLabel作为应用程序的窗口的。

第7行使QLabel标签(label)可见。在创建窗口部件的时候,标签通常都是隐藏的,这就允许我们可以先对其进行设置然后再显示它们,从而避免了窗口部件的闪烁现象。

第8行将应用程序的控制权传递给Qt。此时,程序会进入事件循环状态,这是一种等待模式,程序会等候用户的动作,例如鼠标单击和按键等操作。用户的动作会让可以产生响应的程序生成一些事件(event,也称为"消息"),这里的响应通常就是执行一个或者多个函数。例如,当用户单击窗口部件时,就会产生一个"鼠标按下"事件和一个"鼠标松开"事件。在这方面,图形用户界面应

用程序和常规的批处理程序完全不同，后者通常可以在没有人为干预的情况下自行处理输入、生成结果和终止。

为简单起见，我们没有过多关注在 main()函数末尾处对 QLabel 对象的 delete 操作调用。在如此短小的程序内，这样一点内存泄漏(memory leak)问题无关大局，因为在程序结束时，这部分内存是可以由操作系统重新回收的。

现在是在机器上测试这个程序的时候了。看起来它应当会如图 1.1 所示。首先，需要安装 Qt 4.3.2(或是其后的其他 Qt 4 新发行版)，附录 A 对这一安装过程进行了说明。从现在开始，假定你已经正确地安装了 Qt 4 的一个副本，并且假定已经在 PATH 环境变量中对 Qt 的 bin 目录进行了设置。(在 Windows 操作系统中，这些操作会由 Qt 的安装程序自动完成。)还需要将该程序的源代码保存到 hello.cpp 文件，并把它放进一个名为 hello 的目录中。你可以自行把代码录入到 hello.cpp 文件，也可以从与本书配套的那些例子中复制该文件，它放在 examples/chap01/hello/hello.cpp 文件中。(所有例子都可以从本书的网站中获取，网址是 http://www.informit.com/title/0132354160。)



图 1.1　Linux 上的 Hello 程序

在命令提示符下，进入 hello 目录，输入如下命令，生成一个与平台无关的项目文件 hello.pro：

```
qmake -project
```

然后，输入如下命令，从这个项目文件生成一个与平台相关的 makefile 文件：

```
qmake hello.pro
```

键入 make 命令就可以构建该程序。(在附录 B 中，会给出 qmake 工具更为详细的说明。)要运行该程序，在 Windows 下可以输入 hello，在 UNIX 下可以输入 ./hello，在 Mac OS X 下可以输入 open hello.app。要结束该程序，可直接单击窗口标题栏上的关闭按钮。

如果使用的是 Windows 系统，并且已经安装了 Qt 的开源版和 MinGW 编译器，那么将会看到一个指向 MS-DOS 提示符窗口的快捷键，其中已经正确地创建了使用 Qt 时所需的全部环境变量。如果启动了这个窗口，那么就可以在里面像上面所讲述的那样使用 qmake 命令和 make 命令编译 Qt 应用程序。而由此产生的可执行文件将会保存在应用程序所在目录的 debug 或 release 文件夹中(例如，C:\examples\chap01\hello\release\hello.exe)。

如果使用的是 Microsoft Visual C++ 和商业版的 Qt，则需要用 nmake 命令代替 make 命令。除了这一方法外，还可以通过 hello.pro 文件创建一个 Visual Studio 的工程文件，此时需要输入命令：

```
qmake -tp vc hello.pro
```

然后就可以在 Visual Studio 中编译这个程序了。如果使用的是 Mac OS X 系统中的 Xcode，那么可以使用如下命令来生成一个 Xcode 工程文件：

```
qmake -spec macx-xcode hello.pro
```

在开始进入下一个例子之前，我们一起来做一件有意思的事情：将代码行

```
QLabel *label = new QLabel("Hello Qt!");
```

替换为

```
QLabel *label = new QLabel("<h2><i>Hello</i> "
                           "<font color=red>Qt!</font></h2>");
```

然后重新编译该程序。运行程序时，看起来应当是图 1.2 的样子。正如该例子所显示的那样，通过使用一些简单的 HTML 样式格式，就可以轻松地把 Qt 应用程序的用户接口变得更为丰富多彩。

sudo dpkg--configure -a



图 1.2 具有简单 HTML 样式的标签

1.2 建立连接

第二个例子要说明的是如何响应用户的动作。这个应用程序由一个按钮构成,用户可以单击这个按钮退出程序。除了应用程序的主窗口部件使用的是 QPushButton 而不是 QLabel 之外,这个应用程序的源代码和 Hello 程序的源代码非常相似。同时,我们还会将用户的一个动作(单击按钮)与一段代码连接起来。

这个应用程序的源代码位于本书的例子文件中,文件名是 examples/chap01/quit/quit.cpp。程序的运行效果如图 1.3 所示。以下是该文件所包含的内容:



图 1.3 Quit 应用程序

```
1  #include <QApplication>
2  #include <QPushButton>

3  int main(int argc, char *argv[])
4  {
5      QApplication app(argc, argv);
6      QPushButton *button = new QPushButton("Quit");
7      QObject::connect(button, SIGNAL(clicked()),
8                       &app, SLOT(quit()));
9      button->show();
10     return app.exec();
11 }
```

Qt 的窗口部件通过发射信号(signal)来表明一个用户动作已经发生了或者是一个状态已经改变了[①]。例如,当用户单击 QPushButton 时,该按钮就会发射一个 clicked()信号。信号可以与函数(在这里称为槽,slot)相连接,以便在发射信号时,槽可以得到自动执行。在这个例子中,我们把这个按钮的 clicked()信号与 QApplication 对象的 quit()槽连接起来。宏 SIGNAL()和SLOT()是Qt 语法中的一部分。

现在来构建这个应用程序。假设已经创建了一个包含 quit.cpp 文件的 quit 目录。在 quit 目录中,首先运行 qmake 命令生成它的工程文件,然后再次运行该命令来生成一个 makefile 文件,这两项操作的命令如下:

```
qmake -project
qmake quit.pro
```

现在,就可以编译并运行这个应用程序了。如果单击 Quit 按钮,或者按下了空格键(这样也会按下 Quit 按钮),那么将会退出应用程序。

1.3 窗口部件的布局

这一节将创建一个简单的例子程序,以说明如何用布局(layout)来管理窗口中窗口部件的几何形状,还要说明如何利用信号和槽来同步窗口部件。这个应用程序的运行效果如图 1.4所示,它可以用来询问用户的年龄,而用户可以通过操纵微调框(spin box)或者滑块(slider)来完成年龄的输入。

① Qt 的信号和 UNIX 的信号并不相关,本书中所讨论的信号仅指 Qt 信号。

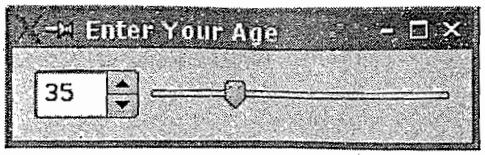


图 1.4　Age 应用程序

这个应用程序由三个窗口部件组成：一个 QSpinBox，一个 QSlider 和一个 QWidget。QWidget 是这个应用程序的主窗口。QSpinBox 和 QSlider 会显示在 QWidget 中，它们都是 QWidget 窗口部件的子对象。换言之，QWidget 窗口部件是 QSpinBox 和 QSlider 的父对象。QWidget 窗口部件自己则没有父对象，因为程序是把它当作顶层窗口的。QWidget 的构造函数以及它的所有子类都会带一个参数 QWidget *，以用来说明谁是它们的父窗口部件。

以下是本应用程序的源代码：

```
1  #include <QApplication>
2  #include <QHBoxLayout>
3  #include <QSlider>
4  #include <QSpinBox>

5  int main(int argc, char *argv[])
6  {
7      QApplication app(argc, argv);

8      QWidget *window = new QWidget;
9      window->setWindowTitle("Enter Your Age");

10     QSpinBox *spinBox = new QSpinBox;
11     QSlider *slider = new QSlider(Qt::Horizontal);
12     spinBox->setRange(0, 130);
13     slider->setRange(0, 130);

14     QObject::connect(spinBox, SIGNAL(valueChanged(int)),
15                      slider, SLOT(setValue(int)));
16     QObject::connect(slider, SIGNAL(valueChanged(int)),
17                      spinBox, SLOT(setValue(int)));
18     spinBox->setValue(35);

19     QHBoxLayout *layout = new QHBoxLayout;
20     layout->addWidget(spinBox);
21     layout->addWidget(slider);
22     window->setLayout(layout);

23     window->show();

24     return app.exec();
25 }
```

第 8 行和第 9 行创建了 QWidget 对象，并把它作为应用程序的主窗口。我们通过调用 setWindowTitle()函数来设置显示在窗口标题栏上的文字。

第 10 行和第 11 行分别创建了一个 QSpinBox 和一个 QSlider，并分别在第 12 行和第 13 行设置了它们的有效范围。我们可以放心地假定用户的最大年龄不会超过 130 岁。本应把这个窗口传递给 QSpinBox 和 QSlider 的构造函数，以说明这两个窗口部件的父对象都是这个窗口，但在这里没有这个必要，因为布局系统将会自行得出这一结果并自动把该窗口设置为微调框和滑块的父对象，下面将很快看到这一点。

从第 14 行到第 17 行，调用了两次 QObject::connect()，这是为了确保能够让微调框和滑块同步，以便它们两个总是可以显示相同的数值。一旦有一个窗口部件的值发生了改变，那么就会发射它的 valueChanged(int)信号，而另一个窗口部件就会用这个新值调用它的 setValue(int)槽。

第 18 行将微调框的值设置为 35。当发生这种情况时，QSpinBox 就会发射 valueChanged(int)信号，其中，int 参数的值是 35。这个参数会被传递给 QSlider 的 setValue(int)槽，它会把这个滑块的值设置为 35。于是，滑块就会发射 valueChanged(int)信号，因为它的值发生了变化，这样就触发了微

调框的 setValue(int)槽。但在这一点上,setValue(int)并不会再发射任何信号,因为微调框的值已经是 35 了。这样就可以避免无限循环的发生。图 1.5 对这种情况进行了图示概述。

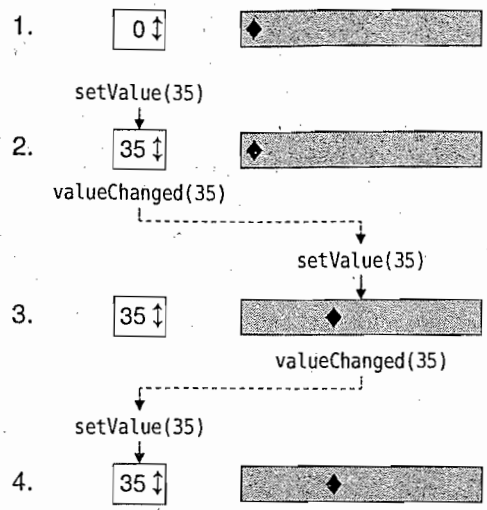


图 1.5 改变一个窗口部件的值会使两个窗口部件都发生变化

窗口部件的风格

到现在为止,我们看到的这些屏幕截图都来自于 Linux,但是 Qt 应用程序在每一个所支持的平台上都可以看起来像本地程序一样(见图 1.6)。Qt 是通过所模拟平台的视觉外观来实现这一点的,而不是对某个特殊平台的封装或者一个工具包中的窗口部件集。

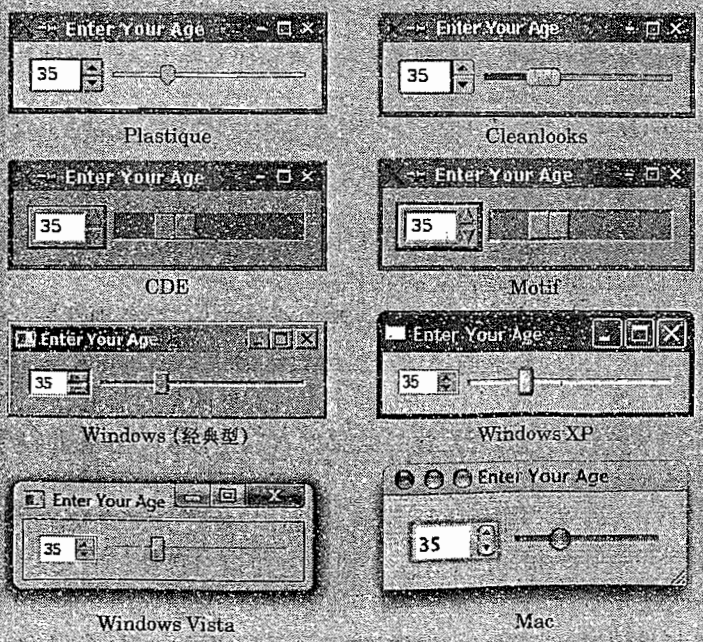


图 1.6 一些预定义风格

运行于KDE下的Qt/X11应用程序的默认风格是Plastique，而运行于GNOME下的应用程序的默认风格是Cleanlooks。这些风格使用了渐变和抗锯齿效果，以用来提供一种时尚的外观(look and feel)。运行Qt应用程序的用户可以通过使用命令行参数-style覆盖原有的默认风格。例如，在X11下，要想使用Motif风格来运行Age应用程序，只需简单地输入以下命令即可：

```
./age -style motif
```

与其他风格不同，Windows XP、Windows Vista和Mac的风格只能在它们的本地平台上有效，因为它们需要依赖平台的主题引擎。

还有另外一种风格QtDotNet，它来自于Qt Solutions模块。创建自定义风格也是可能的，这会在第19章中加以阐述。

在源程序的第19行到第22行，使用了一个布局管理器对微调框和滑块进行布局处理。布局管理器(layout manager)就是一个能够对其所负责窗口部件的尺寸大小和位置进行设置的对象。Qt有三个主要的布局管理器类：

- QHBoxLayout。在水平方向上排列窗口部件，从左到右(在某些文化中则是从右向左)。
- QVBoxLayout。在竖直方向上排列窗口部件，从上到下。
- QGridLayout。把各个窗口部件排列在一个网格中。

第22行的QWidget::setLayout()函数调用会在窗口上安装该布局管理器(见图1.7)。从软件的底层实现上来说，QSpinBox和QSlider会自动“重定义父对象”，它们会成为这个安装了布局的窗口部件的子对象。也正是基于这种原因，当创建一个需要放进某个布局中的窗口部件时，就没有必要为其显式地指定父对象了。

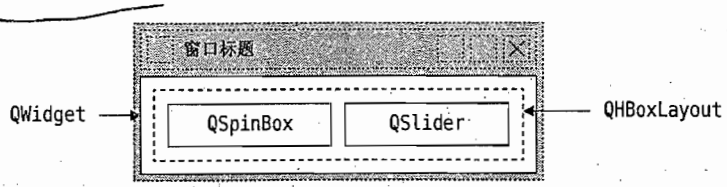


图1.7 Age应用程序中的窗口部件和布局

尽管没有明确地设置任何一个窗口部件的位置或大小，但QSpinBox和QSlider还是能够非常好看地一个挨着一个显示出来。这是因为QHBoxLayout可根据所负责的子对象的需要为它们分配所需的位置和大小。布局管理器使我们从应用程序的各种屏幕位置关系指定的繁杂纷扰中解脱出来，并且它还可以确保窗口尺寸大小发生改变时的平稳性。

Qt中构建用户接口的方法很容易理解并且非常灵活。Qt程序员最常使用的方式是先声明所需的窗口部件，然后再设置它们所应具备的属性。程序员把这些窗口部件添加到布局中，布局会自动设置它们的位置和大小。利用Qt的信号和槽机理，并通过窗口部件之间的连接就可以管理用户的交互行为。

1.4 使用参考文档

由于Qt的参考文档涉及了Qt中的每一个类和函数，所以对任何一名Qt开发人员来说，它都是一个基本工具。本书讲述了Qt的许多类和函数，但是也并不能完全覆盖到Qt中所有的类和函

数,同时也无法对书中所涉及的每个类和函数都提供全部的细节。如果想尽可能多地从 Qt 获益,那么就应当尽快地达到对 Qt 参考文档了如指掌的程度。

在 Qt 的 doc/html 目录下可以找到 HTML 格式的参考文档,并且可以使用任何一种 Web 浏览器来阅读它。也可以使用 Qt 的帮助浏览器 Qt Assistant,它具有强大的查询和索引功能,使用时能够比 Web 浏览器更加快速和容易。

要运行 Qt Assistant,在 Windows 下,可单击"开始"菜单中的"Qt by Trolltech v4.x.y | Assistant"(见图 1.8);在 UNIX 下,可在命令行终端中输入 assistant 命令;在 Mac OS X Finder 中,只需双击 assistant 即可,在主页的"API Reference"小节中的链接提供了浏览 Qt 类的几种不同方式,"All Classes"页面列表会列出 Qt API 中的每一个类,而"Main Classes"页面列表只会列出 Qt 中那些最为常用的类。作为练习,你或许可以去试着查询一下这一章中所使用过的那些类和函数。

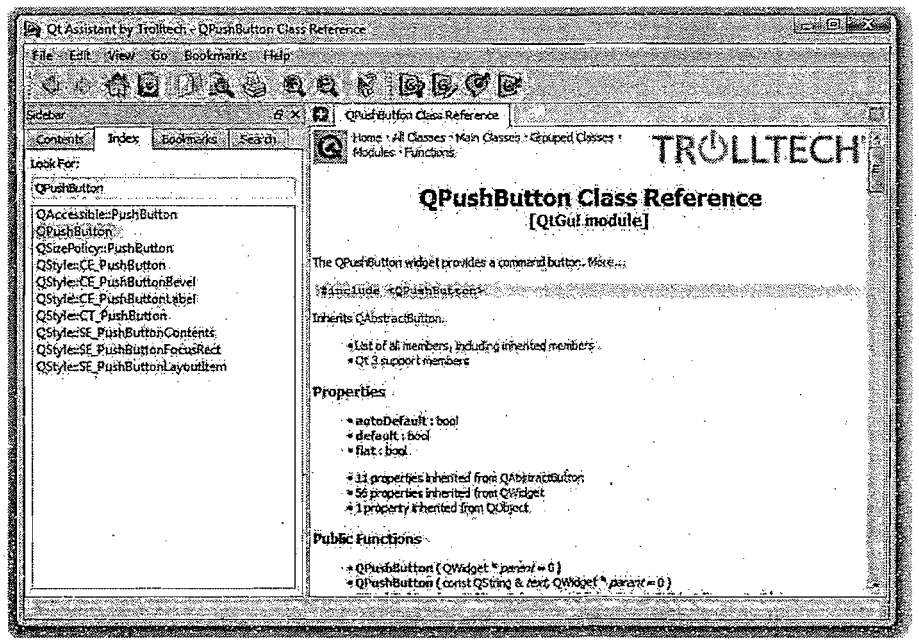


图 1.8 Windows Vista 下 Qt Assistant 中的 Qt 参考文档

需要注意的是,通过继承而得到的函数的文档会显示在它的基类中,例如,QPushButton 就没有它自己的 show()函数,因为它是从 QWidget 那里继承的函数。图 1.9 给出了到目前为止我们所见过的各个类之间的关系。

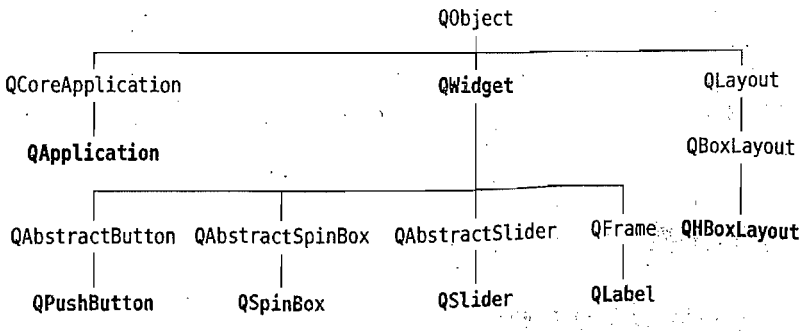


图 1.9 目前为止我们所见过的那些 Qt 类的继承树

可以从 http://doc.trolltech.com 中获取 Qt 的当前版和一些早期版本的在线参考文档。这个网站也选摘了 Qt 季刊(Qt Quarterly)中的一些文章。Qt 季刊是 Qt 程序员的时事通讯,会发送给所有获得 Qt 商业许可协议的人员。

本章介绍了一些重要概念:信号-槽连接和布局,也逐步展示了 Qt 的兼容性和 Qt 完全面向对象的构建方法和窗口部件的使用。如果你浏览了一遍 Qt 的参考文档,那么将会发现一种如何学习使用新窗口部件的统一方法,并且也将发现 Qt 对函数、参数、枚举等变量选名的严谨性,以及在使用 Qt 编程时令人叹服的愉悦感和舒适性。

本书第一部分的随后几章,都建立在本章的基础之上,它们演示了如何创建一个完整的 GUI 应用程序——拥有菜单、工具栏、文档窗口、状态条和对话框,还有与之相应的用于阅读、处理和输出文件的底层功能函数。

第 2 章　创建对话框

这一章讲解如何使用 Qt 创建对话框。对话框为用户提供了许多选项和多种选择，允许用户把选项设置为他们喜欢的变量值并从中做出选择。之所以把它们称为对话框，或者简称为“对话”，是因为它们为用户和应用程序之间提供了一种可以相互“交谈”的交互方式。

绝大多数图形用户界面应用程序都带有一个由菜单栏、工具栏构成的主窗口以及几十个对主窗口进行补充的对话框。当然，也可以创建对话框应用程序，它可以通过执行合适的动作来直接响应用户的选择(例如，一个计算器应用程序)。

本章将首先完全用手写代码的方式创建第一个对话框，以便能够说明是如何完成这项工程的。然后将使用 Qt 的可视化界面设计工具 Qt 设计师(Qt Designer)。使用 Qt 设计师比手写代码要快得多，并且可以使不同的设计测试工作以及稍后对设计的修改工作变得异常轻松。

2.1　子类化 QDialog

第一个例子是完全使用 C++ 编写的一个 Find(查找)对话框，它的运行效果如图 2.1 所示，这将实现一个拥有自主权的对话框。通过这一过程，就可以让对话框拥有自己的信号和槽，成为一个独立的、完备的控件。

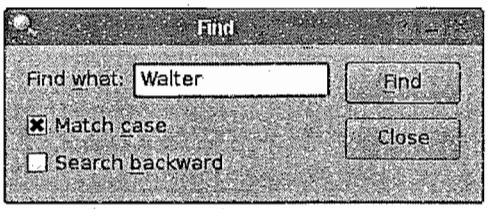


图 2.1　Find 对话框

源代码分别保存在 finddialog.h 和 finddialog.cpp 文件中。首先从 finddialog.h 文件说起：

```
1 #ifndef FINDDIALOG_H
2 #define FINDDIALOG_H

3 #include <QDialog>

4 class QCheckBox;
5 class QLabel;
6 class QLineEdit;
7 class QPushButton;
```

第 1、2 行(以及后面的第 27 行，见下页)能够防止对这个头文件的多重包含。第 3 行包含了 QDialog 的定义，它是 Qt 中对话框的基类。QDialog 从 QWidget 类中派生出来。第 4 行到第 7 行前置声明了一些用于这个对话框实现中的 Qt 类。前置声明(forward declaration)会告诉 C++ 编译程序类的存在，而不用提供类定义中的所有细节(通常放在它自己的头文件中)。关于这一点，将会再简单地多讲一些。

接下来定义 FindDialog，并让它成为 QDialog 的子类：

```
 8 class FindDialog : public QDialog
 9 {
10     Q_OBJECT

11 public:
12     FindDialog(QWidget *parent = 0);
```

对于所有定义了信号和槽的类，在类定义开始处的 Q_OBJECT 宏都是必需的。

FindDialog 的构造函数就是一个典型的 Qt 窗口部件类的定义方式。parent 参数指定了它的父窗口部件。该参数的默认值是一个空指针，意味着该对话框没有父对象。

```
13 signals:
14     void findNext(const QString &str, Qt::CaseSensitivity cs);
15     void findPrevious(const QString &str, Qt::CaseSensitivity cs);
```

signals 部分声明了当用户单击 Find 按钮时对话框所发射的两个信号。如果向前查询(search backward)选项生效，对话框就发射 findPrevious()信号，否则它就发射 findNext()信号。

signals 关键字实际上是一个宏。C++ 预处理器会在编译程序找到它之前把它转换成标准 C++ 代码。Qt::CaseSensitivity 是一个枚举类型，它有 Qt::CaseSensitive 和 Qt::CaseInsensitive 两个取值。

```
16 private slots:
17     void findClicked();
18     void enableFindButton(const QString &text);

19 private:
20     QLabel *label;
21     QLineEdit *lineEdit;
22     QCheckBox *caseCheckBox;
23     QCheckBox *backwardCheckBox;
24     QPushButton *findButton;
25     QPushButton *closeButton;
26 };

27 #endif
```

在这个类的 private 段声明了两个槽。为了实现这两个槽，几乎需要访问这个对话框的所有子窗口部件，所以也保留了指向它们的指针。关键字 slots 就像 signals 一样也是一个宏，也可以扩展成 C++ 编译程序可以处理的一种结构形式。

对于这些私有变量，我们使用了它们的类前置声明。这是可行的，因为它们都是指针，而且没有必要在头文件中就去访问它们，因而编译程序就无须这些类的完整定义。我们没有包含与这几个类相关的头文件(如 <QCheckBox> 、<QLabel> ，等等)，而是使用了一些前置声明，这可以使编译过程更快一些。

现在看一下 finddialog.cpp，其中包含了对 FindDialog 类的实现：

```
1 #include <QtGui>

2 #include "finddialog.h"
```

首先，需要包含 <QtGui> ，该头文件包含了 Qt GUI 类的定义。Qt 由数个模块构成，每个模块都有自己的类库。最为重要的模块有 QtCore、QtGui、QtNetwork、QtOpenGL、QtScript、QtSql、QtSvg 和 QtXml。其中，在 <QtGui> 头文件中为构成 QtCore 和 QtGui 组成部分的所有类进行了定义。在程序中包含这个头文件，就能够使我们省去在每个类中分别包含的麻烦。

在 filedialog.h 文件中，本可以仅简单地添加一个 <QtGui> 包含即可，而不用包含 <QDialog> 和使用 QCheckBox、QLabel、QLineEdit 和 QPushButton 的前置声明。然而，在一个头文件中再包含一个那么大的头文件着实不是一种好的编程风格，尤其对于比较大的工程项目更是如此。

```
3  FindDialog::FindDialog(QWidget *parent)
4      : QDialog(parent)
5  {
6      label = new QLabel(tr("Find &what:"));
7      lineEdit = new QLineEdit;
8      label->setBuddy(lineEdit);

9      caseCheckBox = new QCheckBox(tr("Match &case"));
10     backwardCheckBox = new QCheckBox(tr("Search &backward"));

11     findButton = new QPushButton(tr("&Find"));
12     findButton->setDefault(true);
13     findButton->setEnabled(false);

14     closeButton = new QPushButton(tr("Close"));
```

在第 4 行,把 parent 参数传递给了基类的构造函数。然后,创建了子窗口部件。在字符串周围的 tr()函数调用是把它们翻译成其他语言的标记。在每个 QObject 对象以及包含有 Q_OBJECT 宏的子类中都有这个函数的声明。尽管也许并没有将你的应用程序立刻翻译成其他语言的打算,但是在每一个用户可见的字符串周围使用 tr()函数还是一个很不错的习惯。在第 18 章中将对翻译 Qt 应用程序进行详细讲述。

在这些字符串中,使用了表示"与"操作的符号"&"来表示快捷键。例如,第 11 行创建了一个 Find 按钮,用户可在那些支持快捷键的平台下通过按下 Alt + F 快捷键来激活它。符号"&"可以用来控制焦点:在第 6 行创建了一个带有快捷键(Alt + W)的标签,而在第 8 行设置了行编辑器作为标签的伙伴。所谓"伙伴"(buddy),就是一个窗口部件,它可以在按下标签的快捷键时接收焦点(focus)。所以当用户按下 Alt + W(该标签的快捷键)时,焦点就会移动到这个行编辑器(该标签的伙伴)上。

在第 12 行,通过调用 setDefault(true)让 Find 按钮成为对话框的默认按钮。默认按钮(default button)就是当用户按下 Enter 键时能够按下对应的按钮。在第 13 行,禁用了 Find 按钮。当禁用一个窗口部件时,它通常会显示为灰色,并且不能和用户发生交互操作。

```
15     connect(lineEdit, SIGNAL(textChanged(const QString &)),
16             this, SLOT(enableFindButton(const QString &)));
17     connect(findButton, SIGNAL(clicked()),
18             this, SLOT(findClicked()));
19     connect(closeButton, SIGNAL(clicked()),
20             this, SLOT(close()));
```

只要行编辑器中的文本发生变化,就会调用私有槽 enableFindButton(const QString &)。当用户单击 Find 按钮时,会调用 findClicked()私有槽。而当用户单击 Close 时,对话框会关闭。close()槽是从 QWidget 中继承而来的,并且它的默认行为就是把窗口部件从用户的视野中隐藏起来(而无须将其删除)。稍后将会看到 enableFindButton()槽和 findClicked()槽的代码。

由于 QObject 是 FindDialog 的父对象之一,所以可以省略 connect()函数前面的 QObject::前缀。

```
21     QHBoxLayout *topLeftLayout = new QHBoxLayout;
22     topLeftLayout->addWidget(label);
23     topLeftLayout->addWidget(lineEdit);

24     QVBoxLayout *leftLayout = new QVBoxLayout;
25     leftLayout->addLayout(topLeftLayout);
26     leftLayout->addWidget(caseCheckBox);
27     leftLayout->addWidget(backwardCheckBox);

28     QVBoxLayout *rightLayout = new QVBoxLayout;
29     rightLayout->addWidget(findButton);
30     rightLayout->addWidget(closeButton);
31     rightLayout->addStretch();

32     QHBoxLayout *mainLayout = new QHBoxLayout;
```

```
33      mainLayout->addLayout(leftLayout);
34      mainLayout->addLayout(rightLayout);
35      setLayout(mainLayout);
```

接下来，使用布局管理器摆放这些子窗口部件。布局中既可以包含多个窗口部件，也可以包含其他子布局。通过 QHBoxLayout、QVBoxLayout 和 QGridLayout 这三个布局的不同嵌套组合，就可能构建出相当复杂的对话框。

如图 2.2 所示，对于 Find 对话框，使用了两个 QHBoxLayout 布局和两个 QVBoxLayout 布局。外面的布局是主布局，通过第 35 行代码将其安装在 FindDialog 中，并且由其负责对话框的整个区域。其他三个布局则作为子布局。图 2.2 中右下角的小“弹簧”是一个分隔符（或者称为“伸展器”）。用它来占据 Find 按钮和 Close 按钮所余下的空白区域，这样可以确保这些按钮完全占用它们所在布局的上部空间。

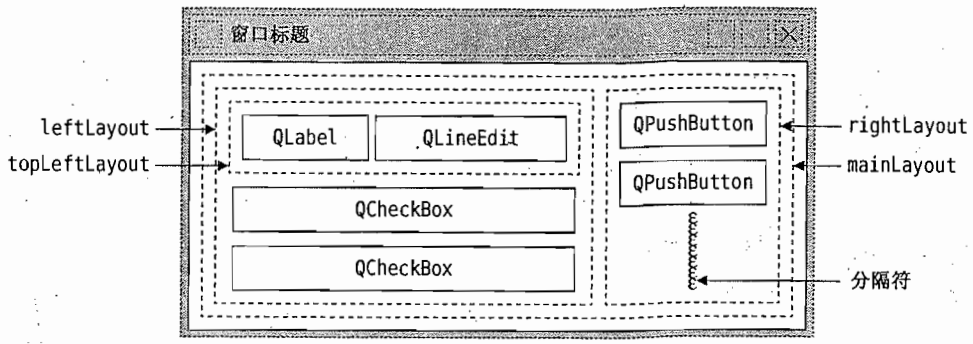


图 2.2　Find 对话框的布局

布局管理器类的一个精妙之处在于它们不是窗口部件。相反，它们派生自 QLayout，因而也就是进一步派生自 QObject。在图 2.2 中，窗口部件用实线轮廓来表示，布局用点线来表示，这样就能够很好地区分窗口部件和布局。在一个运行的应用程序中，布局是不可见的。

当将子布局对象添加到父布局对象中时（第 25、33 和 34 行），子布局对象就会自动重定义自己的父对象。也就是说，当将主布局装到对话框中去时（第 35 行），它就会成为对话框的子对象了，于是它的所有子窗口部件都会重定义自己的父对象，从而变成对话框中的子对象。图 2.3 给出了父子层次关系的最终结果。

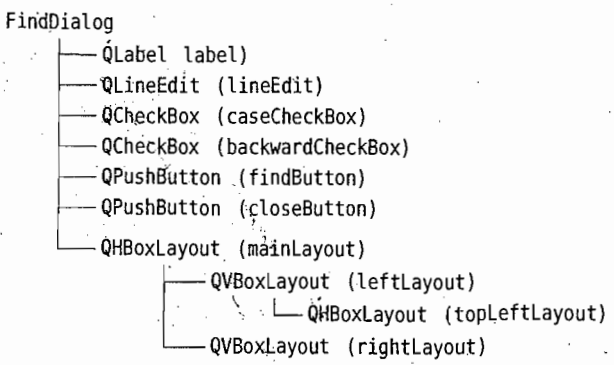


图 2.3　Find 对话框中的父子关系

```
36      setWindowTitle(tr("Find"));
37      setFixedHeight(sizeHint().height());
38  }
```

最后,设置了显示在对话框标题栏上的标题的内容,并让窗口具有一个固定的高度,这是因为在对话框的垂直方向上再没有其他窗口部件可以去占用所多出的空间了。QWidget::sizeHint()函数可以返回一个窗口部件所"理想"的尺寸大小。

这样,就完成了对 FindDialog 对话框构造函数的分析。由于在创建这个对话框中的窗口部件和布局时使用的是 new,所以需要写一个能够调用 delete 的析构函数,以便可以删除所创建的每一个窗口部件和布局。但是这样做并不是必需的,因为 Qt 会在删除父对象的时候自动删除其所属的所有子对象,也就会删除 FindDialog 中作为其子孙的所有子窗口部件和子布局。

现在来看一下这个对话框中所用到的槽:

```
39 void FindDialog::findClicked()
40 {
41     QString text = lineEdit->text();
42     Qt::CaseSensitivity cs =
43             caseCheckBox->isChecked() ? Qt::CaseSensitive
44                                       : Qt::CaseInsensitive;
45     if (backwardCheckBox->isChecked()) {
46         emit findPrevious(text, cs);
47     } else {
48         emit findNext(text, cs);
49     }
50 }

51 void FindDialog::enableFindButton(const QString &text)
52 {
53     findButton->setEnabled(!text.isEmpty());
54 }
```

当用户单击 Find 按钮时,就会调用 findClicked()槽。而该槽将会发射 findPrevious()或 findNext()信号,这取决于 Search backward 选项的取值。emit 是 Qt 中的关键字,像其他 Qt 扩展一样,它也会被 C++ 预处理器转换成标准的 C++ 代码。

只要用户改变了行编辑器中的文本,就会调用 enableFindButton()槽。如果在行编辑器中有文本,该槽就会启用 Find 按钮,否则它就会禁用 Find 按钮。

利用这两个槽就完成了这个对话框的功能。现在,可以创建一个 main.cpp 文件来测试一下这个 FindDialog 窗口部件。

```
1 #include <QApplication>

2 #include "finddialog.h"

3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     FindDialog *dialog = new FindDialog;
7     dialog->show();
8     return app.exec();
9 }
```

为了编译这个程序,还像以前一样运行 qmake。由于 FindDialog 类的定义包含 Q_OBJECT宏,因而由 qmake 生成的 makefile 将会自动包含一些运行 moc 的规则,moc 就是指 Qt 的元对象编译器,即 meta-object compiler。(会在下一节介绍 Qt 的元对象系统。)

为了使 moc 能够正常运行,必须把类定义从实现文件中分离出来并放到头文件中。由 moc 生成的代码会包含这个头文件,并且会添加一些特殊的 C++ 代码。

必须对使用了 Q_OBJECT宏的类运行 moc。因为 qmake 会自动在 makefile 中添加这些必要的规则,所以这并不成问题。但是如果忘记了使用 qmake 重新生成 makefile 文件,并且也没有重新运行 moc,那么连接程序就会报错,指出你声明了一些函数但是却没有实现它们。这些信息可能是相当

不明确的。GCC 会生成像这样的出错信息：

```
finddialog.o: In function `FindDialog::tr(char const*, char const*)':
/usr/lib/qt/src/corelib/global/qglobal.h:1430: undefined reference to
`FindDialog::staticMetaObject'
```

Visual C++ 输出的出错信息可能是这样的：

```
finddialog.obj : error LNK2001: unresolved external symbol
"public:~virtual int __thiscall MyClass::qt_metacall(enum QMetaObject
::Call,int,void * *)"
```

如果曾经遇到过这种情况，那么请重新运行 qmake 以生成新的 makefile 文件，然后再重新构建该应用程序。

现在来运行该程序。如果在你的系统上能够显示快捷键，那么可以检验一下快捷键 Alt + W、Alt + C、Alt + B 和 Alt + F 是不是触发了正确的行为。可以通过敲击键盘上的 Tab 键来遍历这些窗口部件。默认的 Tab 键顺序就是创建窗口部件时的顺序。要改变这个键顺序，可以使用 QWidget::setTabOrder()函数。

提供一种合理的 Tab 键顺序和键盘快捷键可以确保不愿(或者不能)使用鼠标的用户能够充分享受应用程序所提供的全部功能。完全通过键盘控制应用程序也深受快速输入人员的赞赏。

在第 3 章，将在一个真实的应用程序中使用 Find 对话框，并且将会把 findPrevious()信号和 findNext()信号与一些槽连接到一起。

2.2　深入介绍信号和槽

信号和槽机制是 Qt 编程的基础。它可以让应用程序编程人员把这些互不了解的对象绑定在一起。前面，已经把一些信号和槽连接在了一起，也声明了自己的信号和槽，还实现了自己的槽，并且还发射了自己的信号。让我们再花一点时间，来进一步深入地了解这个机制。

槽和普通的 C++ 成员函数几乎是一样的——可以是虚函数；可以被重载；可以是公有的、保护的或者私有的，并且也可以被其他 C++ 成员函数直接调用；还有，它们的参数可以是任意类型。唯一的不同是：槽还可以和信号连接在一起，在这种情况下，每当发射这个信号的时候，就会自动调用这个槽。

connect()语句看起来会是如下的样子：

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

这里的 *sender* 和 *receiver* 是指向 QObject 的指针，*signal* 和 *slot* 是不带参数的函数名。实际上，SIGNAL()宏和 SLOT()宏会把它们的参数转换成相应的字符串。

到目前为止，在已经看到的实例中，我们已经把不同的信号和不同的槽连接在了一起。但这里还需要考虑一些其他的可能性。

- **一个信号可以连接多个槽**

```
connect(slider, SIGNAL(valueChanged(int)),
        spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int)));
```

在发射这个信号的时候，会以不确定的顺序一个接一个地调用这些槽。

● 多个信号可以连接同一个槽

```
connect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```

无论发射的是哪一个信号,都会调用这个槽。

● 一个信号可以与另外一个信号相连接

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));
```

当发射第一个信号时,也会发射第二个信号。除此之外,信号与信号之间的连接和信号与槽之间的连接是难以区分的。

● 连接可以被移除

```
disconnect(lcd, SIGNAL(overflow()),
           this, SLOT(handleMathError()));
```

这种情况较少用到,因为当删除对象时,Qt 会自动移除和这个对象相关的所有连接。

要把信号成功连接到槽(或者连接到另外一个信号),它们的参数必须具有相同的顺序和相同的类型:

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(processReply(int, const QString &)));
```

这里有个例外,如果信号的参数比它所连接的槽的参数多,那么多余的参数将会被简单地忽略掉:

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(checkErrorCode(int)));
```

如果参数类型不匹配,或者如果信号或槽不存在,则当应用程序使用调试模式构建后,Qt 会在运行时发出警告。与之相类似的是,如果在信号和槽的名字中包含了参数名,Qt 也会发出警告。

到现在为止,我们仅仅在窗口部件之间使用了信号和槽。但是这种机制本身是在 QObject 中实现的,并不只局限于图形用户界面编程中。这种机制可以用于任何 QObject 的子类中:

```
class Employee : public QObject
{
    Q_OBJECT

public:
    Employee() { mySalary = 0; }

    int salary() const { return mySalary; }

public slots:
    void setSalary(int newSalary);

signals:
    void salaryChanged(int newSalary);

private:
    int mySalary;
};
void Employee::setSalary(int newSalary)
{
    if (newSalary != mySalary) {
        mySalary = newSalary;
        emit salaryChanged(mySalary);
    }
}
```

注意一下 setSalary()槽是如何工作的。只有在 newSalary != mySalary 的时候,才发射 salaryChanged()信号。这样可以确保循环连接不会导致无限循环。

Qt 的元对象系统

Qt 的主要成就之一就是使用了一种机制对 C++ 进行了扩展，并且使用这种机制创建了独立的软件组件。这些组件可以绑定在一起，但任何一个组件对于它所要连接的组件的情况事先都一无所知。

这种机制称为元对象系统(meta-object system)，它提供了关键的两项技术：信号-槽以及内省(introspection)。内省功能对于实现信号和槽是必需的，并且允许应用程序的开发人员在运行时获得有关 QObject 子类的“元信息”(meta-information)，包括一个含有对象的类名以及它所支持的信号和槽的列表。这一机制也支持属性(广泛用于 Qt 设计师中)和文本翻译(用于国际化)，并且它也为 QtScript 模块奠定了基础。从 Qt 4.2 开始，可以动态添加属性，这一特性将会在第 19 章和第 22 章中付诸实施。

标准 C++ 没有对 Qt 的元对象系统所需要的动态元信息提供支持。Qt 通过提供一个独立的 moc 工具解决了这个问题，moc 解析 Q_OBJECT 类的定义并且通过 C++ 函数来提供可供使用的信息。由于 moc 使用纯 C++ 来实现它的所有功能，所以 Qt 的元对象系统可以在任意 C++ 编译器上工作。

这一机制是这样工作的：

- Q_OBJECT 宏声明了在每一个 QObject 子类中必须实现的一些内省函数：metaObject()、tr()、qt_metacall()，以及其他一些函数。
- Qt 的 moc 工具生成了用于由 Q_OBJECT 声明的所有函数和所有信号的实现。
- 像 connect()和 disconnect()这样的 QObject 的成员函数使用这些内省函数来完成它们的工作。

由于所有这些工作都是由 qmake、moc 和 QObject 自动处理的，所以很少需要再去考虑这些事情。但是如果你对此充满好奇心，那么也可以阅读一下有关 QMetaObject 类的文档和由 moc 生成的 C++ 源代码文件，可以从中看出这些实现工作是如何进行的。

2.3　快速设计对话框

Qt 的设计初衷就是为了能够直观并且友好地进行手工编码，并且对于程序员来说，纯粹通过编写 C++ 源代码来开发整个 Qt 应用程序并不稀奇。尽管如此，许多程序员还是喜欢使用可视化的方法来设计窗体，因为他们发现使用可视化方式会比手工编码显得更自然、更快速，并且也希望能够通过可视化方法，对那些手工编码所设计的窗体，进行更快速、更容易的测试和修改。

Qt 设计师(Qt Designer)为程序员们提供了可供使用的新选择，它提供一种可视化的设计能力。Qt 设计师可用于开发应用程序中的所有或部分窗体。使用 Qt 设计师所创建的窗体最终仍旧是 C++ 代码，因此，可把 Qt 设计师看作是一个传统的工具集，并且不会对编译器强加其他特殊要求。

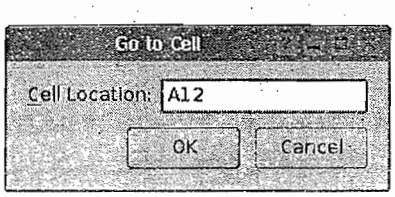


图 2.4　Go to Cell 对话框

在这一节，将使用 Qt 设计师来创建如图 2.4 所示的 Go to Cell 对话框。并且无论是使用手工编码还是使用 Qt 设计师，在创建对话框时总是要包含以下这几个相同的基本步骤：

1. 创建并初始化子窗口部件。
2. 把子窗口部件放到布局中。
3. 设置 Tab 键顺序。
4. 建立信号-槽之间的连接。
5. 实现对话框中的自定义槽。

要启动 Qt 设计师,在 Windows 下,可单击“开始”菜单中的 Qt by Trolltech v4.x.y→Designer;在 UNIX 下,在命令行中输入“designer”;在 Mac OS X Finder 中,直接双击 designer。当 Qt 设计师开始运行后,它会弹出一个多种模板的列表。单击 Widget 模板,然后再单击 Create。(“Dialog with Buttons Bottom”模板看起来可能更具诱惑力,但是对于这个例子来说,为了能够看到是如何完成 OK 和 Cancel 按钮的,所以需要采用手工方式来创建它们。)现在应该会看到一个名为“Untitled”的窗口。

默认情况下,Qt 设计师的用户界面由多个顶级窗口构成。如果你更喜欢像图 2.5 所示的那种多文档(MDI)界面风格,即只有一个顶级窗口和多个子窗口构成的界面,则可以单击 Edit→Preferences,然后将用户界面模式设置为 Docked Window(停靠窗口)即可。

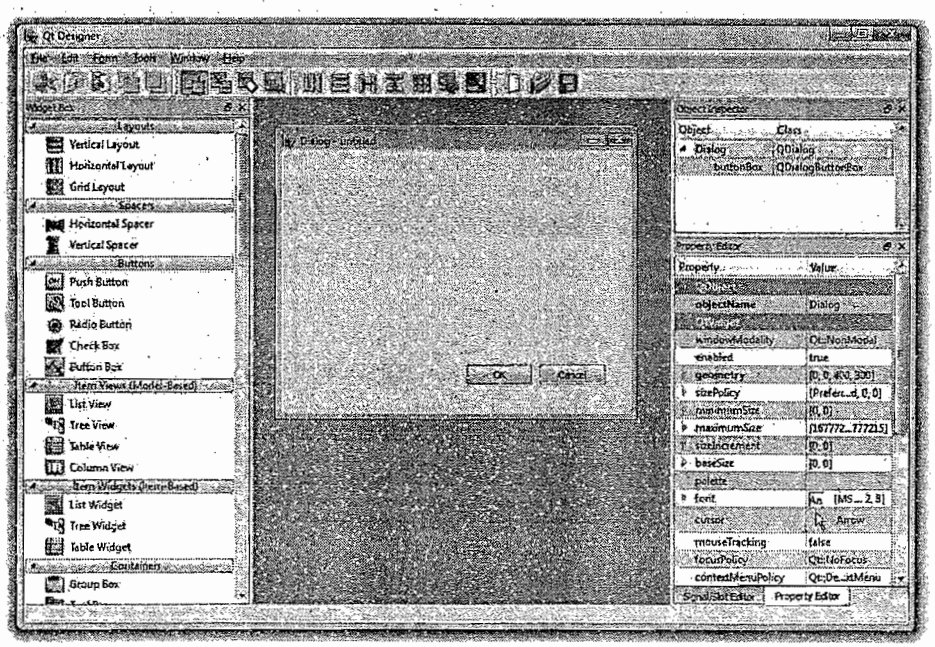


图 2.5 Windows Vista 中显示为停靠窗口模式的 Qt 设计师

第一步是创建子窗口部件并且把它们放置到窗体中。创建一个标签、一个行编辑器、一个水平分隔符和两个按钮。对于这里的每一项,可先从 Qt 设计师的窗口部件工具箱中拖拽其名字或者图标并将其放到窗体中的大概位置。在 Qt 设计师中,分隔符会显示为一个蓝色的弹簧,但在最终结果的窗体中它是不可见的。

现在,向上拖动窗体的底部使它变短一些,这样将会产生一个类似于图 2.6 的窗体。不要在窗体上为确定这些项的位置而花费太多的时间,会在稍后使用 Qt 的布局管理器,它可以把这些项摆放得恰到好处。

使用 Qt 设计师的属性编辑器可以设置每一个窗口部件中的属性:

1. 单击文本标签。确保此时 objectName 的属性是“label”,那么就可以将它的 text 属性设置成“&Cell Location:”。

2. 单击行编辑器。确保 objectName 属性是“lineEdit”。
3. 单击第一个按钮。将它的 objectName 属性设置成“okButton”，将它的 enabled 属性设置成“false”，将它的 text 属性设置成“OK”，并且把它的 default 属性设置成“true”。
4. 单击第二个按钮。将它的 objectName 属性设置成“cancelButton”，并且将它的 text 属性设置成“Cancel”。
5. 单击这个窗体中空白的地方，选中窗体本身。将 objectName 属性设置成“GoToCellDialog”，并且将它的 windowTitle 属性设置成“Go to Cell”。

现在，除了文本标签，所有的窗口部件看起来都很不错，文本标签仍显示为“&Cell Location:”。单击 Edit→Edit Buddies 进入一种允许设置窗口部件伙伴(buddy)的特殊模式。然后，单击这个标签并把红色箭头拖到行编辑器上，释放鼠标按键。现在标签看起来应该显示为“Cell Location:”，如图 2.7 所示，同时，它还会把行编辑器看成是自己的伙伴。单击 Edit→Edit Widgets 离开伙伴设置模式。

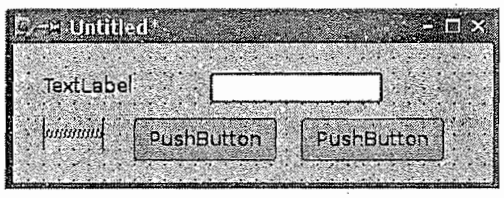


图 2.6　带一些窗口部件的窗体

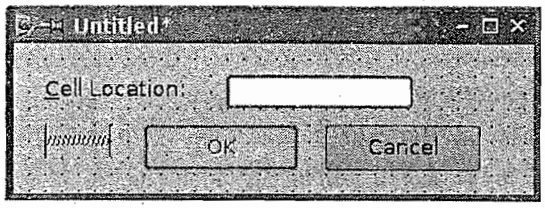


图 2.7　带属性设置的窗体

下一步是在窗体中摆放这些窗口部件，步骤如下：

1. 单击“Cell Location:”标签并且当单击与之相邻的行编辑器时按下 Shift 键，这样就可以同时选择它们。单击 Form→Lay Out Horizontally。
2. 单击分隔符，然后在单击窗体的 OK 按钮和 Cancel 按钮时一直按下 Shift 键。单击 Form→Lay Out Horizontally。
3. 单击窗体中的空白，取消对所有已选中项的选择，然后单击 Form→Lay Out Vertically。
4. 单击 Form→Adjust Size，重新把窗体的大小定义为最佳形式。

在窗体上出现的红线就是已经创建的布局，如图 2.8 所示。但是在窗体运行的时候，它们是绝不会出现的。

现在，单击 Edit→Edit Tab Order。在每一个可以接受焦点的窗口部件上，都会出现一个带蓝色矩形的数字，如图 2.9 所示。按照你所希望的接受焦点的顺序单击每一个窗口部件，然后单击 Edit→Edit Widgets，离开 Tab 键顺序设置模式。

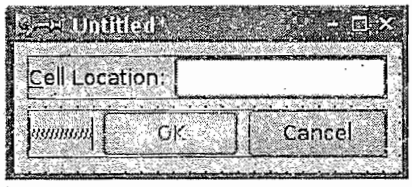


图 2.8　带布局的窗体

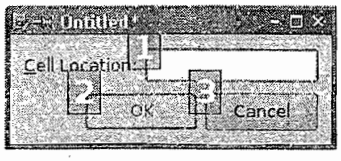


图 2.9　设置窗体的 Tab 键顺序

要预览这个对话框，可单击 Form→Preview 菜单选项。通过重复按下 Tab 键来检查对话框 Tab 键的顺序。使用窗体标题栏上的 Close 按钮，可以关闭对话框。

把对话框保存到 gotocell 目录下，另存为 gotocelldialog.ui，然后使用一个纯文本编辑器在同一目录下创建一个 main.cpp 文件，内容如下：

```
#include <QApplication>
#include <QDialog>

#include "ui_gotocelldialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    Ui::GoToCellDialog ui;
    QDialog *dialog = new QDialog;
    ui.setupUi(dialog);
    dialog->show();

    return app.exec();
}
```

现在运行 qmake,生成一个 .pro 文件和一个 makefile 文件(命令分别是:qmake -project; qmake gotocell.pro)。qmake 工具非常智能,它可以自动检测到用户界面文件 gotocelldialog.ui 并且可以生成适当的 makefile 规则来调用 Qt 的用户界面编译器(user interface compiler,uic)。uic 工具会将 gotocelldialog.ui 文件转换成 C++ 并且将转换结果存储在 ui_gotocelldialog.h 文件中。

所生成的 ui_gotocelldialog.h 文件中包含了类 Ui::GoToCellDialog 的定义,该类是一个与 gotocelldialog.ui 文件等价的 C++ 文件。这个类声明了一些成员变量,它们存储着窗体中的子窗口部件和子布局,以及用于初始化窗体的 setupUi()函数。生成的类看起来如下所示:

```
class Ui::GoToCellDialog
{
public:
    QLabel *label;
    QLineEdit *lineEdit;
    QSpacerItem *spacerItem;
    QPushButton *okButton;
    QPushButton *cancelButton;
    ...

    void setupUi(QWidget *widget) {
        ...
    }
};
```

生成的类没有任何基类。当在 main.cpp 文件中使用该窗体时,可以创建一个 QDialog 对象,然后把它传递给 setupUi()函数。

如果现在运行该程序,对话框也可以工作,但它并没有正确地实现所想要的那些功能:

- OK 按钮总是失效的。
- Cancel 按钮什么也做不了。
- 行编辑器可以接受任何文本,而不是只能接受有效的单元格位置坐标。

通过写一些代码,就可以让对话框具有适当的功能。最为简捷的做法是创建一个新类,让该类同时从 QDialog 和 Ui::GoToCellDialog 中继承出来,并且由它来实现那些缺失的功能(从而也证明了这句话:通过简单地增加另外一个间接层就可以解决软件的任何问题)。命名惯例是:将该类与 uic 所生成的类具有相同的名字,只是没有 Ui::前缀而已。

使用文本编辑器,创建一个名为 gotocelldialog.h 的文件,其中所包含的代码如下所示:

```
#ifndef GOTOCELLDIALOG_H
#define GOTOCELLDIALOG_H

#include <QDialog>

#include "ui_gotocelldialog.h"
```

```
class GoToCellDialog : public QDialog, public Ui::GoToCellDialog
{
    Q_OBJECT

public:
    GoToCellDialog(QWidget *parent = 0);

private slots:
    void on_lineEdit_textChanged();
};

#endif
```

在这里，使用了 public 继承，这是因为我们想在该对话框的外面访问该对话框的窗口部件。包含在 gotocelldialog.cpp 文件中的实现代码如下所示：

```
#include <QtGui>

#include "gotocelldialog.h"

GoToCellDialog::GoToCellDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);

    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));

    connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
    connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
}

void GoToCellDialog::on_lineEdit_textChanged()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
```

在构造函数中，调用 setupUi()函数来初始化窗体。正是由于使用了多重继承关系，可以直接访问 Ui::GoToCellDialog 中的成员。创建了用户接口后，setupUi()函数还会自动将那些符合 on_objectName_signalName()命名惯例的任意槽与相应的 objectName 的 signalName()信号连接到一起。在这个例子中，这就意味着 setupUi()函数将建立如下所示的信号-槽连接关系：

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SLOT(on_lineEdit_textChanged()));
```

同样还是在构造函数中，设置了一个检验器来限制输入的范围。Qt 提供了三个内置检验器类：QIntValidator、QDoubleValidator 和 QRegExpValidator。在这里使用检验器类 QRegExpValidator，让它带一个正则表达式“[A-Za-z][1-9][0-9]{0,2}”，它的意思是：允许一个大写或者小写的字母，后面跟着一个范围为 1～9 的数字，后面再跟 0 个、1 个或 2 个 0～9 的数字。（对于正则表达式的介绍，请查看参考文档中的 QRegExp 类。）

通过把 this 传递给 QRegExpValidator 的构造函数，使它成为 GoToCellDialog 对象的一个子对象。这样，以后就不用担心有关删除 QRegExpValidator 的事情了：当删除它的父对象时，它也会被自动删除。

Qt 的父-子对象机制是在 QObject 中实现的。当利用一个父对象创建一个子对象（一个窗口部件，一个检验器，或是任意的其他类型）时，父对象会把这个子对象添加到自己的子对象列表中。当删除这个父对象时，它会遍历子对象列表并且删除每一个子对象。然后，这些子对象再去删除它们自己所包含的每个子对象。如此反复递归调用，直至清空所有子对象为止。这种父-子对象机制可在很大程度上简化内存管理工作，降低内存泄漏的风险。需要明确删除的对象是那些使用 new 创建的并且没有父对象的对象。并且，如果在删除一个父对象之前先删除了它的子对象，Qt 会自动地从它的父对象的子对象列表中将其移除。

对于窗口部件,父对象还有另外一层含义:子窗口部件会显示在它的父对象所在的区域中。当删除这个父窗口部件时,不仅子对象会从内存中消失,而且它也会在屏幕上消失。

在构造函数的最后部分,我们将 OK 按钮连接到 QDialog 的 accept()槽,将 Cancel 按钮连接到 reject()槽。这两个槽都可以关闭对话框,但 accept()槽可以将对话框返回的结果变量设置为 QDialog::Accepted(其值等于 1),而 reject()槽会把对话框的值设置为 QDialog::Rejected(其值等于 0)。当使用这个对话框的时候,可以利用这个结果变量判断用户是否单击了 OK 按钮,从而执行相应的动作。

根据行编辑器中是否包含了有效的单元格位置坐标,on_lineEdit_textChanged()槽可以启用或者禁用 OK 按钮。QLineEdit::hasAcceptableInput()会使用在构造函数中设置的检验器来判断行编辑器中内容的有效性。

这样,就完成了这个对话框。现在可以通过重写 main.cpp 文件来使用这个对话框:

```
#include <QApplication>

#include "gotocelldialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    GoToCellDialog *dialog = new GoToCellDialog;
    dialog->show();
    return app.exec();
}
```

使用 qmake -project 命令重新生成 gotocell.pro 文件(因为已经在工程中添加了源文件),使用 qmake gotocell.pro 命令更新 makefile 文件,然后再次构建并运行应用程序。在行编辑器中输入"A12",这时可以注意到 OK 按钮已经变为启用了。尝试输入一些随机文本来看看检验器是如何完成它的工作的。单击 Cancel 按钮关闭对话框。

这个对话框工作得很好,但对于 Mac OS X 用户,这些按钮却显得不够圆润。在前面采用了单独添加每个按钮的方法,这是为了可以让我们看出是如何完成这些步骤的,但我们本来的确是应当使用 QDialogButtonBox 的,它是一个可以容纳给定的按钮的窗口部件,可以让那些按钮以正确方式呈现在应用程序所运行的平台上,如图 2.10 所示。

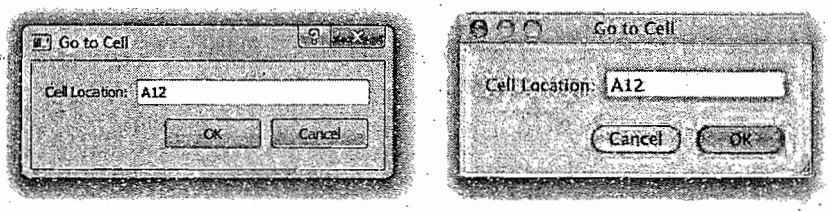


图 2.10 Windows Vista 和 Mac OS X 上的 Go to Cell 对话框

要使用 QDialogButtonBox 来制作这个对话框,必须同时修改设计过程和上述代码。在 Qt 设计师中,一共需要 4 步:

1. 单击窗体(不是任何窗口部件或者布局),然后单击 Form→Break Layout。
2. 单击并删除 OK 按钮、Cancel 按钮、水平分隔符以及(现在为空的)水平布局。
3. 在窗体上拖放一个"按钮盒"(Button Box),放在标签和行编辑器单元的下方。
4. 单击窗体,然后单击 Form→Lay Out Vertically。

如果只打算修改设计,比如修改对话框的布局和窗口部件的属性等,那么只要重新构建应用

程序即可。但在这里是移除了一些窗口部件并且添加了一个新的窗口部件，所以在这种情况下，通常还必须对代码进行修改。

我们所必须做的修改都在 gotocelldialog.cpp 文件中。这里给出的是其构造函数的新版本：

```
GoToCellDialog::GoToCellDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(false);

    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));

    connect(buttonBox, SIGNAL(accepted()), this, SLOT(accept()));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
}
```

在前一版本中，一开始在 Qt 设计师中禁用了 OK 按钮。但是在使用 QDialogButtonBox 之后就不能那样做了，因而可以在代码中调用 setupUi()之后，再立即禁用 OK 按钮，这样也可以达到同样的效果。类 QDialogButtonBox 有一组标准按钮的枚举值，并且可以利用这一点来访问这些特殊的按钮，本例就是访问 OK 按钮。

非常方便的是，Qt 设计师对于 QDialogButtonBox 的默认名称就是 buttonBox。双方的连接会从按钮盒而不是从按钮自己创建出来。在单击一个带 AcceptRole 的按钮时，就会发射 accepted()信号，这一点与单击一个带 RejectRole 的按钮会发射 rejected()信号的情况相似。默认情况下，标准的 QDialogButtonBox::Ok 按钮具有 AcceptRole 属性，而标准的 QDialogButtonBox::Cancel 按钮具有 RejectRole 属性。

还需要在 on_lineEdit_textChanged()槽中做一处修改：

```
void GoToCellDialog::on_lineEdit_textChanged()
{
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(
            lineEdit->hasAcceptableInput());
}
```

与之前的唯一不同之处在于不是对存储为成员变量的特殊按钮进行引用，而是直接去访问按钮盒中的 OK 按钮。

使用 Qt 设计师的一个好处就在于它为程序员在修改自己设计的窗体时提供了很大的自由，并且不必再强迫自己去修改源代码。当完全通过手写 C++ 代码开发窗体时，对窗体设计的修改将会相当耗时。利用 Qt 设计师，由于 uic 会自动为那些发生了改变的窗体重新生成源代码，所以就不会再浪费时间了。对话框的用户交互界面会被保存为 .ui 文件(一种基于 XML 的文件格式)，而通过对 uic 所生成的类进行子类化，就可以实现自定义的函数功能。

2.4　改变形状的对话框

我们已经看到了如何创建对话框，无论何时使用它们，这些对话框永远只会显示出一些相同的窗口部件。在某些情况下，人们非常希望能够提供一些可以改变形状的对话框。最常见的可改变形状的对话框有两种：扩展对话框(extension dialog)和多页对话框(multi-page dialog)。在 Qt 中，不论是纯粹使用代码还是使用 Qt 设计师，都可以实现这两种对话框。

扩展对话框通常只显示简单的外观，但是它还有一个切换按钮(toggle button)，可以让用户在对话框的简单外观和扩展外观之间来回切换。扩展对话框通常用于试图同时满足普通用户和高级用户需要的应用程序中，这种应用程序通常会隐藏那些高级选项，除非用户明确要求看到它们。在这一节中，将使用 Qt 设计师来创建如图 2.11 所示的扩展对话框。

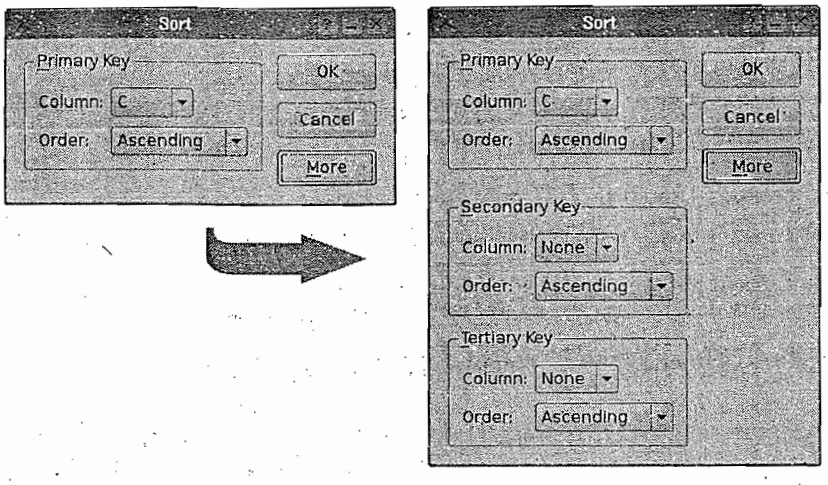


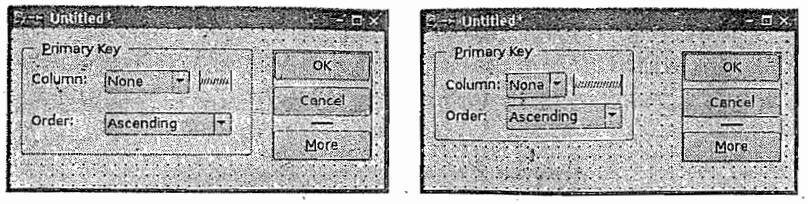
图 2.11 具有简单外观和扩展外观的 Sort 对话框

这个对话框是一个用于电子制表软件应用程序的排序对话框(Sort 对话框),在这个对话框中,用户可以选择一列或多列进行排序。在这个简单外观中,允许用户输入一个单一的排序键,而在扩展外观下,还额外提供了两个排序键。More 按钮允许用户在简单外观和扩展外观之间切换。

我们将在 Qt 设计师中创建这个对话框的扩展外观,并且在运行时根据需要隐藏排序的第二键和第三键。这个窗口部件看起来有些复杂,但在 Qt 设计师中可以轻而易举地完成它。简单的诀窍是首先完成主键部分,然后再复制并且粘贴两次就可以获得第二键和第三键所需的内容。

1. 单击 File→New Form,并选择"Dialog without Buttons"模板。
2. 创建 OK 按钮并把它拖放到窗体的右上角。将它的 objectName 修改为"okButton",并将它的 default 属性设置为"true"。
3. 创建 Cancel 按钮并把它拖放到 OK 按钮的下方。将它的 objectName 修改为"cancelButton"。
4. 创建一个垂直分隔符并把它拖放到 Cancel 按钮的下方,然后再创建一个 More 按钮。并将它放在垂直分隔符的下方。将 More 按钮的 objectName 修改为"moreButton",text 属性设置成"&More",checkable 属性设置为"true"。
5. 单击 OK 按钮,按下 Shift 键后再单击 Cancel 按钮、垂直分隔符和 More 按钮,然后单击 Form→Lay Out Vertically。
6. 创建一个群组框、两个标签、两个组合框以及一个水平分隔符,然后把它们放到窗体上的任意位置。
7. 拖动群组框的右下角使它变大一些。然后,把其他窗口部件移到群组框中,并且按照如图 2.12(a)所示的那样把它们放置到适当位置。
8. 拖动第二个组合框的右边缘,使它的宽度大约为第一个组合框的两倍。
9. 将群组框的 title 属性设置为"&Primary Key",第一个标签的 text 属性设置为"Column:",第二个标签的 text 属性设置为"Order:"。
10. 右键单击第一个组合框,从 Qt 设计师弹出的上下文菜单的组合框编辑器中选择 Edit Items。用文本"None"创建一个项。
11. 右键单击第二个组合框并且同样选择 Edit Items。创建一个"Ascending"项和一个"Descending"项。

12. 单击群组框，然后单击 Form→Lay Out in a Grid。再次单击群组框，并且单击 Form→Adjust Size。此时将会产生一个如图 2.12(b)所示的布局。



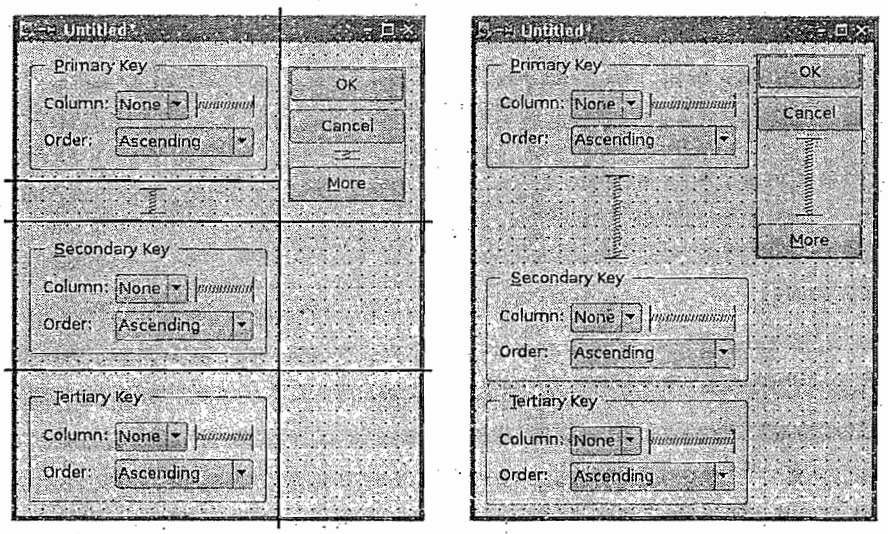
(a) 没有布局的群组框　　(b) 带布局的群组框

图 2.12　将群组框的子对象摆放到网格中

如果没能生成你所希望的那种布局效果，或者是不小心做错了，那么总是可以随时先通过单击 Edit→Undo 或 Form→Break Layout，然后再重新放置这些要摆放的窗口部件，最后再试着对它们重新布局，直到满意为止。

现在来添加其他两个群组框：Secondary Key 和 Tertiary Key：

1. 让对话框窗口足够高，以便可以容纳另外两个部分。
2. 按下 Ctrl 键(在 Mac 中按下 Alt 键)，然后单击并拖动 Primary Key 群组框，这样就可以在原群组框(以及它所包含的所有组件)的上方复制出一个新的群组框。仍旧按下 Ctrl 键(或 Alt 键)，把复制的这个群组框拖动到原群组框的下方。重复以上步骤，就可以生成第三个群组框，然后把它拖动到第二个群组框的下方。
3. 将它们的 title 属性分别修改为“&Secondary Key”和“&Tertiary Key”。
4. 创建一个垂直分隔符，并且把它放到 Primary Key 群组框和 Secondary Key 群组框的中间。
5. 把这些窗口部件按图 2.13(a)所示的那样排列成网格状。



(a) 布局前　　(b) 布局后

图 2.13　把窗体的各个子对象摆放到网格中

6. 单击窗体，取消对任意选中窗口部件的选择，然后单击 Form→Lay Out in a Grid。现在，向上和向左拖动窗体的右下角，以便让窗体变得尽可能地小。现在，窗体应该和图 2.13(b)中显示的一样了。
7. 把两个垂直分隔符的 sizeHint 属性设置为[20,0]。

最终的网格布局是4行2列,一共有8个单元格。Primary Key 群组框、最左边的垂直分隔符、Secondary Key 群组框和 Tertiary Key 群组框各占一个单独的单元格。包含 OK、Cancel 和 More 按钮的垂直布局占用了两个单元格。最后,会在对话框的右下角剩下两个空白单元格。如果你做出来的对话框不是这样,那么请撤销布局,重新放置窗口部件的位置,然后再重新试试。

把这个窗体重命名为"SortDialog",并且把它的标题修改为"Sort"。根据图 2.14 修改各个子窗口部件的名称。

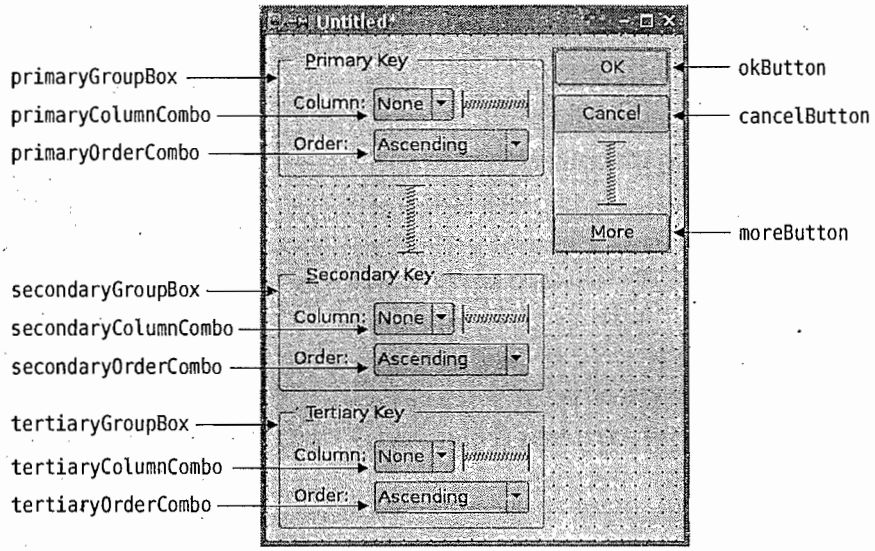


图 2.14 重新命名窗体中的各个窗口部件

单击 Edit→Edit Tab Order,从窗体的最上面到最下面依次单击每个组合框,然后单击窗体右侧的 OK、Cancel 和 More 按钮。单击 Edit→Edit Widgets 离开 Tab 键顺序设置模式。

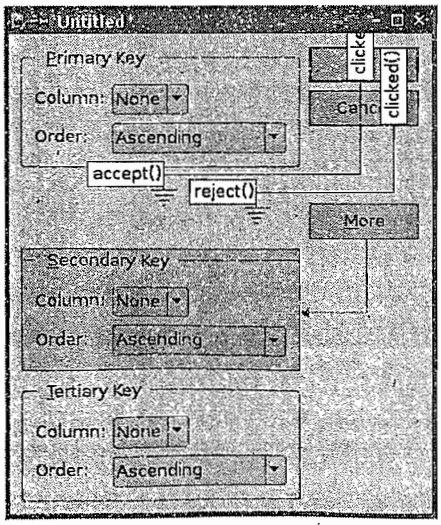


图 2.15 连接窗体的各个部件

现在,窗体已经设计完成,可以开始着手设置一些信号-槽的连接来实现窗体的功能了。Qt 设计师允许我们在构成同一窗体的不同部分内的窗口部件之间建立连接。我们需要建立两个连接。

单击 Edit→Edit Signals/Slots,进入 Qt 设计师的设置连接模式。窗体中各个窗口部件之间的连接用蓝色箭头表示,如图 2.15 所示,并且它们也会同时在 Qt 设计师的 signal/slot 编辑器窗口中显示出来。要在两个窗口部件之间建立连接,可以单击作为发射器的窗口部件并且拖动所产生的红色箭头线到作为接收器的窗口部件上,然后松开鼠标按键。这时会弹出一个对话框,可以从中选择建立连接的信号和槽。

要建立的第一个连接位于 okButton 按钮和窗体的 accept()槽之间。把从 okButton 按钮开始的红色箭头线拖动到窗体的空白区域,然后松开按键,这样会弹出如图 2.16 所示的设置连接对话框(Configure Connection dialog)。从该对话框中选择 clicked()作为信号,选择 accept()作为槽,然后单击 OK 按钮。

对于第二个连接，把从 cancelButton 按钮开始的红色箭头线拖动到窗体的空白区域，然后在设置连接对话框中连接按钮的 clicked()信号和窗体的 reject()槽。

要建立的第三个连接位于 moreButton 按钮和 secondaryGroupBox 群组框之间。在这两个窗口部件之间拖动红色箭头线，然后选择 toggled(bool)作为信号，选择 setVisible(bool)作为槽。默认情况下，setVisible(bool)槽不会显示在 Qt 设计师的槽列表中，但如果选中了“Show all signals and slots”选项，就可以看到这个槽了。

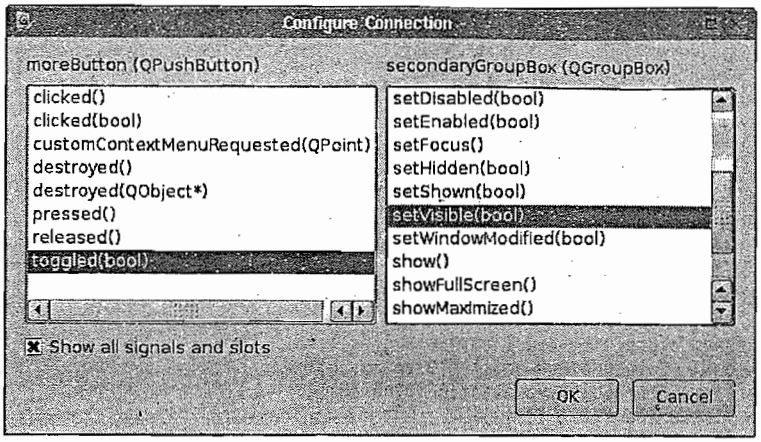


图 2.16　Qt 设计师的连接编辑器

第四个也是最后一个要建立的连接是 moreButton 按钮的 toggled(bool)信号和 tertiaryGroupBox 群组框的 setVisible(bool)槽之间的连接。这些连接一旦完成，就可以单击 Edit→Edit Widgets 而离开创建连接模式。

将这个对话框保存在 sort 目录中，文件名为 sortdialog.ui。要给这个窗体添加代码，同样将使用在前一节的 Go to Cell 对话框设计中已经使用过的多重继承的方法。

首先，用如下内容创建一个 sortdialog.h 文件：

```
#ifndef SORTDIALOG_H
#define SORTDIALOG_H

#include <QDialog>

#include "ui_sortdialog.h"

class SortDialog : public QDialog, public Ui::SortDialog
{
    Q_OBJECT

public:
    SortDialog(QWidget *parent = 0);

    void setColumnRange(QChar first, QChar last);
};

#endif
```

然后再创建 sortdialog.cpp 文件：

```
1 #include <QtGui>

2 #include "sortdialog.h"

3 SortDialog::SortDialog(QWidget *parent)
```

```
4      : QDialog(parent)
5  {
6      setupUi(this);
7
7      secondaryGroupBox->hide();
8      tertiaryGroupBox->hide();
9      layout()->setSizeConstraint(QLayout::SetFixedSize);
10
10     setColumnRange('A', 'Z');
11 }
12 void SortDialog::setColumnRange(QChar first, QChar last)
13 {
14     primaryColumnCombo->clear();
15     secondaryColumnCombo->clear();
16     tertiaryColumnCombo->clear();
17
17     secondaryColumnCombo->addItem(tr("None"));
18     tertiaryColumnCombo->addItem(tr("None"));
19     primaryColumnCombo->setMinimumSize(
20             secondaryColumnCombo->sizeHint());
21
21     QChar ch = first;
22     while (ch <= last) {
23         primaryColumnCombo->addItem(QString(ch));
24         secondaryColumnCombo->addItem(QString(ch));
25         tertiaryColumnCombo->addItem(QString(ch));
26         ch = ch.unicode() + 1;
27     }
28 }
```

构造函数隐藏了对话框的第二键和第三键这两个部分。它也把有关窗体布局的 sizeConstraint 属性设置为 QLayout::SetFixedSize,这样会使用户不能再重新修改这个对话框窗体的大小。这样一来,布局就会负责对话框重新定义大小的职责,并且也会在显示或者隐藏子窗口部件的时候自动重新定义这个对话框的大小,从而可以确保对话框总是能以最佳的尺寸显示出来。

setColumnRange()槽根据电子制表软件中选择的列初始化了这些组合框的内容。在(可选的)第二键和第三键的组合框选项中插入了一个“None”选项。

第 19 行和第 20 行给出了布局中的一个特殊习惯用语。QWidget::sizeHint()函数可以返回布局系统试图认同的“理想”大小。这也解释了为什么不同的窗口部件或者具有不同内容的类似窗口部件通常会被布局系统分配给不同的尺寸大小。对于这些组合框,这里指的是第二键组合框和第三键组合框,由于它们包含了一个“None”选项,所以它们要比只包含了一个单字符项目的主键组合框显得宽一些。为了避免这种不一致性,需要把主键组合框的最小大小设置成第二键组合框的理想大小。

这里是一个用于测试效果的 main()函数,它首先设置了列的范围为从“C”到“F”,然后再显示这个对话框:

```
#include <QApplication>

#include "sortdialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    SortDialog *dialog = new SortDialog;
    dialog->setColumnRange('C', 'F');
    dialog->show();
    return app.exec();
}
```

这样就完成了这个扩展对话框。就像这个实例所显示的那样，设计一个扩展对话框并不比设计一个简单对话框难：所需要的就是一个切换按钮、一些信号-槽连接以及一个不可以改变尺寸大小的布局。在实际的应用程序中，控制扩展对话框的按钮通常会在只显示了基本对话框时显示为 Advanced≫，而在显示了扩展对话框时才显示为 Advanced ≪。这在 Qt 中实现起来非常容易，只需在单击这个按钮时调用 QPushButton 中的 setText()函数即可完成这一功能。

在 Qt 中，无论是使用手工编码的方式还是使用 Qt 设计师，都可以轻松地创建另一种常用的可以改变形状的对话框：多页对话框。可以通过多种不同的方式创建这种对话框：

- QTabWidget 的用法就像它自己的名字一样。它提供了一个可以控制内置 QStackedWidge 的 Tab 栏。
- QListWidget 和 QStackedWidget 可以一起使用，将 QListWidget::currentRowChanged()信号与 QStackedWidget::setCurrentIndex()槽连接，然后再利用 QListWidget 的当前项就可以确定应该显示 QStackedWidget 中的哪一页。
- 与上述 QListWidget 的用法相似，也可以将 QTreeWidget 和 QStackedWidget 一起使用。

第 6 章将讲解 QStackedWidget 类。

2.5　动态对话框

动态对话框(dynamic dialog)就是在程序运行时使用的从 Qt 设计师的 .ui 文件创建而来的那些对话框。动态对话框不需要通过 uic 把 .ui 文件转换成 C++ 代码，相反，它是在程序运行的时候使用 QUiLoader 类载入该文件的，就像下面这种方式：

```
QUiLoader uiLoader;
QFile file("sortdialog.ui");
QWidget *sortDialog = uiLoader.load(&file);
if (sortDialog) {
    ...
}
```

可以使用 QObject::findChild<T>()来访问这个窗体中的各个子窗口部件：

```
QComboBox *primaryColumnCombo =
        sortDialog->findChild<QComboBox *>("primaryColumnCombo");
if (primaryColumnCombo) {
    ...
}
```

这里的 findChild<T>()函数是一个模板成员函数，它可以返回与给定的名字和类型相匹配的子对象。由于受编译器的制约，还不能在 MSVC 6 中使用该函数。如果需要使用 MSVC 6 编译器，那么可以通过调用全局函数 qFindChild<T>()来代替该函数，这个全局函数同样也可以完全相同的方式工作。

QUiLoader 类放在一个独立的库中。为了在 Qt 应用程序中使用 QUiLoader，必须在这个应用程序的 .pro 文件中加入这一行内容：

```
CONFIG += uitools
```

动态对话框使不重新编译应用程序而可以改变窗体布局的做法成为可能。动态对话框也同样可用于创建小型终端应用程序，这些程序只有一个内置的前端窗体，并且只是在需要的时候才会去创建所有的其他窗体。

2.6 内置的窗口部件类和对话框类

Qt 提供了一整套内置的窗口部件和常用对话框,这可以满足绝大多数情况。在这一节,几乎给出了它们所有的屏幕截图。会在稍后提供那些少量的特殊窗口部件:第 3 章会讲到用于主窗口的那些窗口部件,如 QMenuBar、QToolBar 和 QStatusBar 等;第 6 章会讲到与布局相关的那些窗口部件,如 QSplitter 和 QScrollArea 等。在本书提供的实例中将会用到绝大多数内置窗口部件和对话框。所有这些窗口部件都会使用 Plastique 风格显示在从图 2.17 到图 2.26 的屏幕截图中。

如图 2.17 所示,Qt 提供了 4 种类型的按钮:QPushButton、QToolButton、QCheckBox 和 QRadioButton。最常使用的就是 QPushButton 和 QToolButton,当单击时,它们就能够发起一个动作,但它们也可以具有像切换按钮(按钮单击一次被按下,再单击一次会还原)一样的行为。复选框 QCheckBox 可用于打开/关闭单独的那些选项,而单选按钮 QRadioButton 通常用于需要互斥条件的地方。

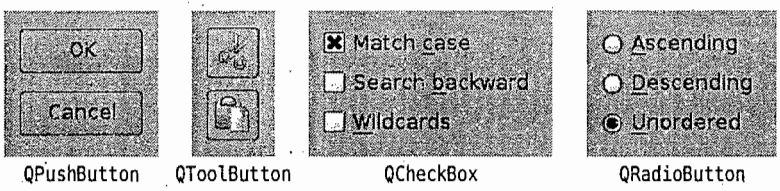


图 2.17 Qt 的按钮窗口部件

Qt 的容器窗口部件是一种可以包含其他窗口部件的窗口部件。图 2.18 和图 2.19 给出了这些容器窗口部件。QFrame 也可用于它自身,这只是为了绘制一些直线,它也可以用作许多其他窗口部件的基类,如 QToolBox 和 QLabel 等。

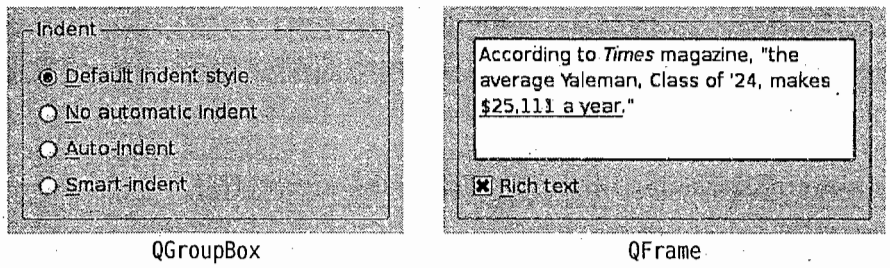
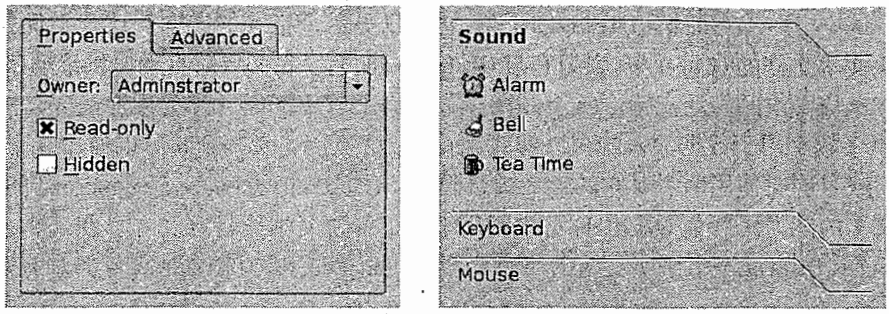


图 2.18 Qt 的单页容器窗口部件

QTabWidget 和 QToolBox 是多页窗口部件。在多页窗口部件中,每一页都是一个子窗口部件,并从 0 开始编号这些页。对于一个 QTabWidget,它的每个 Tab 标签的形状和位置都可以进行设置。

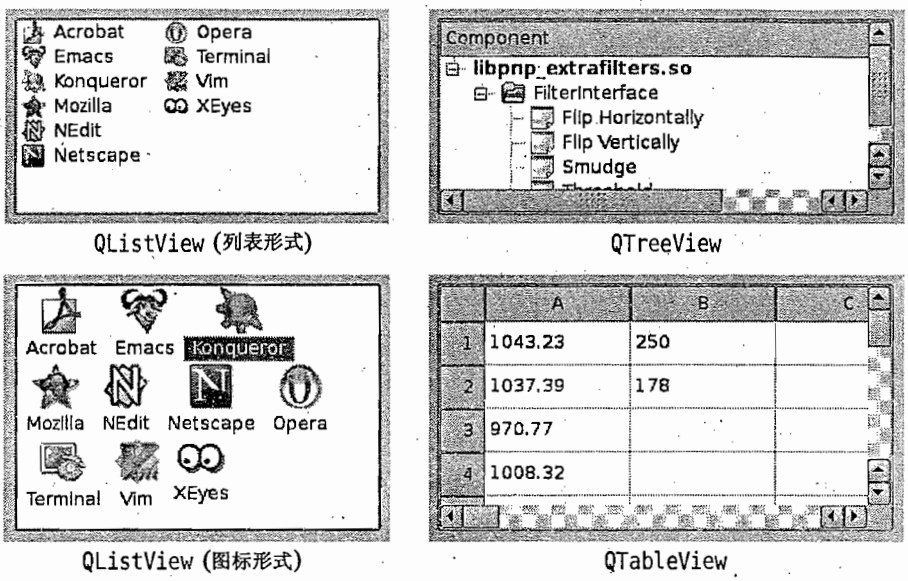
如图 2.20 所示,为处理较大的数据量,这些项视图已经进行了优化,并且会经常使用它们的滚动条(scroll bar)。滚动条机制是在 QAbstractScrollArea 中实现的,它是所有项视图和其他类型的可滚动窗口部件的基类。

Qt 库含有一个富文本引擎(rich text engine),它可用于格式化文本的显示和编辑。该引擎支持字体规范、文本对齐、列表、表格、图片和超文本链接等。可以通过编程的方式一个元素一个元素地生成富文本文档,或者也可以通过所提供的 HTML 格式的文本来生成富文本文档。至于该引擎所支持的 HTML 标记和 CSS 属性的详细说明,请参见文档 http://doc.trolltech.com/4.3/richtext-html-subset.html。



QTabWidget　　QToolBox

图 2.19　Qt 的多页容器窗口部件

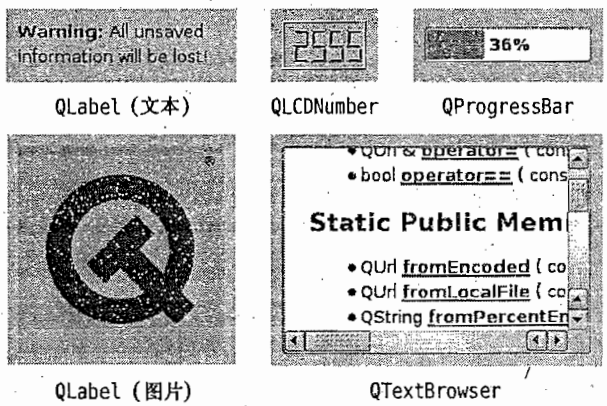


QListView (列表形式)　　QTreeView

QListView (图标形式)　　QTableView

图 2.20　Qt 的项视图窗口部件

如图 2.21 所示，Qt 提供了一些纯粹用于显示信息的窗口部件。QLabel 是这些窗口部件中最重要的一个，并且它也可以用来显示普通文本、HTML 和图片。



QLabel (文本)　　QLCDNumber　　QProgressBar

QLabel (图片)　　QTextBrowser

图 2.21　Qt 的显示窗口部件

QTextBrowser 是一个只读型 QTextEdit 子类,它可以显示带格式的文本。这个类优先用于处理大型格式化文本文档的 QLabel 中,因为它与 QLabel 不同,它会在必要时自动提供滚动条,同时还提供了键盘和鼠标导航的广泛支持。Qt 4.3 助手就是使用 QTextBrowser 来为用户呈现文档的。

Qt 提供了数个用于数据输入的窗口部件,如图 2.22 所示。QLineEdit 可以使用一个输入掩码、一个检验器或者同时使用两者对它的输入进行限定。QTextEdit 是 QAbstractScrollArea 的子类,具有处理大量文本的能力。一个 QTextEdit 可设置用于编辑普通文本或者富文本。在编辑富文本的时候,它可以显示 Qt 富文本引擎所支持的所有元素。QLineEdit 和 QTextEdit 两者都对剪贴板提供完美支持。

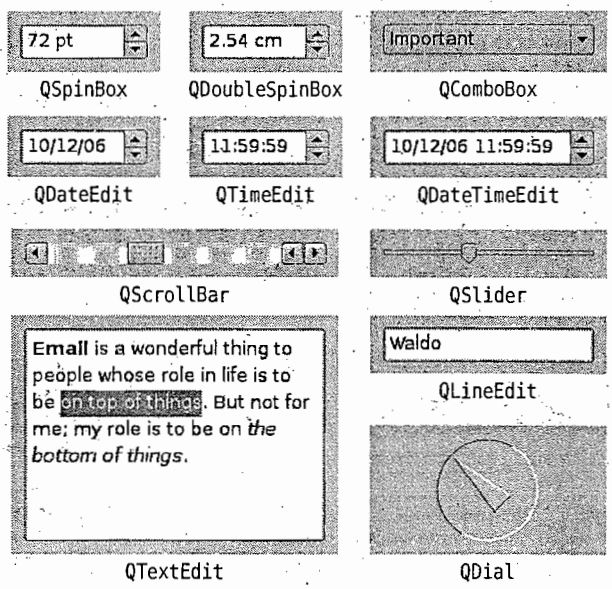


图 2.22 Qt 的输入窗口部件

如图 2.23 所示,Qt 提供了一个通用消息框和一个可以记住它所显示的消息内容的错误对话框。可以使用 QProgressDialog 或者使用图 2.21 中显示的 QProgressBar 来对那些非常耗时的操作进度进行指示。当用户只需要输入一行文本或者一个数字的时候,使用 QInputDialog 会显得非常方便。

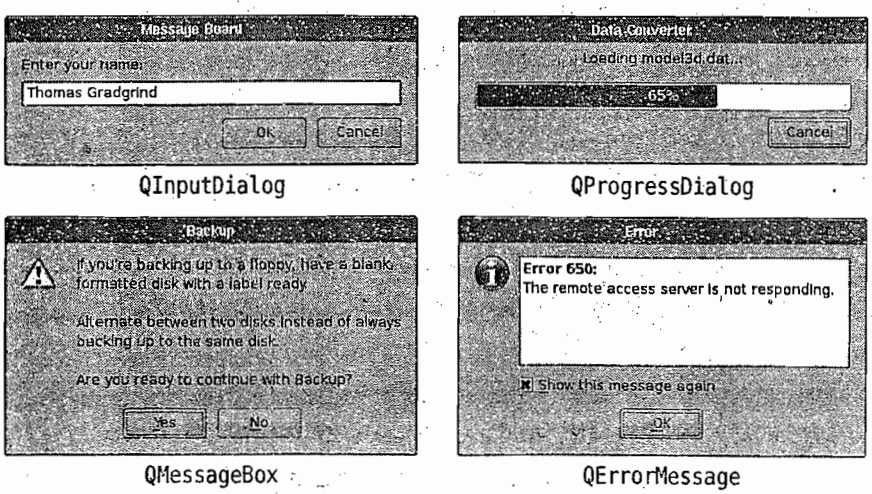
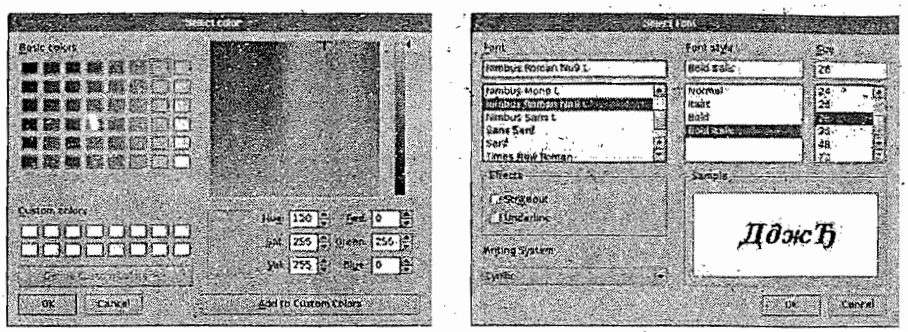


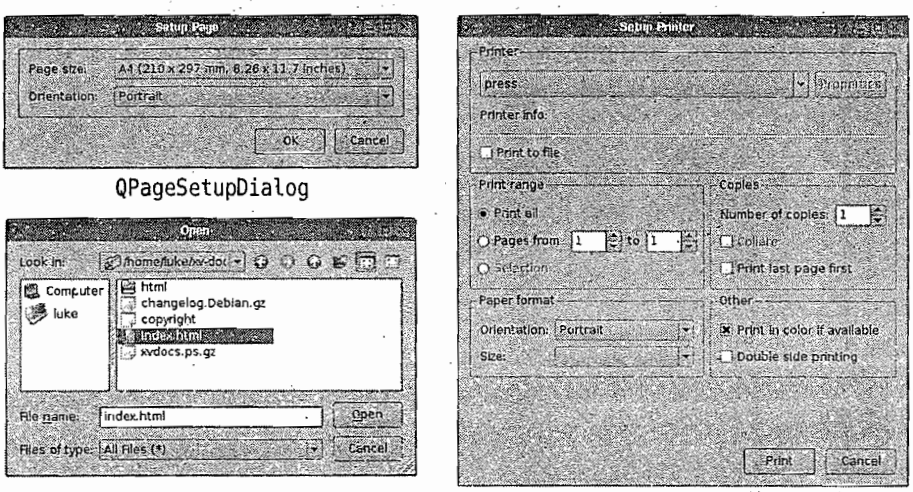
图 2.23 Qt 的反馈对话框

Qt 提供了一套标准的通用对话框，这样可以让用户很容易地选择颜色、字体、文件或者文档打印。图 2.24 和图 2.25 显示了这些对话框。



QColorDialog　　QFontDialog

图 2.24　Qt 的颜色对话框和字体对话框



QPageSetupDialog

QFileDialog　　QPrintDialog

图 2.25　Qt 的文件对话框和打印对话框

在 Windows 和 Mac OS X 上，Qt 有可能会使用本地系统的对话框，而不是它自己的通用对话框。颜色的选取也可以使用 Qt Solutions 的某个颜色选择窗口部件来完成，而字体也可以使用内置的 QFontComboBox 来选择。

最后，QWizard 为生成向导（wizard，在 Mac OS X 上也称为助手）提供了一个基本框架。对于那些用户可能会难于理解的复杂或者不常见的工作，向导会非常有用。图 2.26 给出了使用向导的一个例子。

内置窗口部件和常用对话框为用户提供了很多可以直接使用的功能。通过设置窗口部件的属性，或者是通过把信号和槽连接起来并在槽里实现自定义的行为，通常就可以满足许多更为复杂的需求。

如果 Qt 所提供的窗口部件或者常用对话框没有一个合适，那么可以从 Qt Solutions，或者从商业或非商业的第三方软件中找到一个可用的。Qt Solutions 提供了许多额外的窗口部件，包括各种颜色选择器、一个手轮控制器、许多饼状图菜单以及属性浏览器等，还有一个复制对话框。

在某些情况下，你可能希望手动创建一个自定义窗口部件。Qt 使这种工作变得很简单，并且

自定义窗口部件也可以像 Qt 的内置窗口部件一样获得与平台无关的所有相同绘制功能。自定义窗口部件甚至可以集成到 Qt 设计师中,这样就可以像使用 Qt 的内置窗口部件一样来使用它们。第 5 章将讲述如何创建自定义窗口部件。

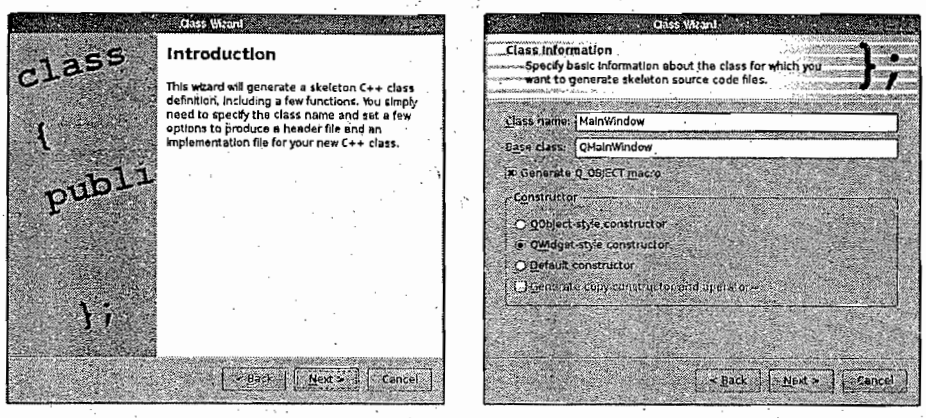


图 2.26 Qt 的 QWizard 对话框

第3章　创建主窗口

这一章讲解如何使用 Qt 创建主窗口。在本章的最后部分，你将能够创建一个应用程序的完整用户界面，包括菜单、工具栏、状态栏以及应用程序所需的足够多的对话框。

应用程序的主窗口提供了用于构建应用程序用户界面的框架。如图 3.1 所示的 Spreadsheet（电子制表）应用程序的主窗口将构成本章的基础。这个 Spreadsheet 应用程序使用了在第 2 章中创建的三个对话框：Find、Go to Cell 和 Sort。

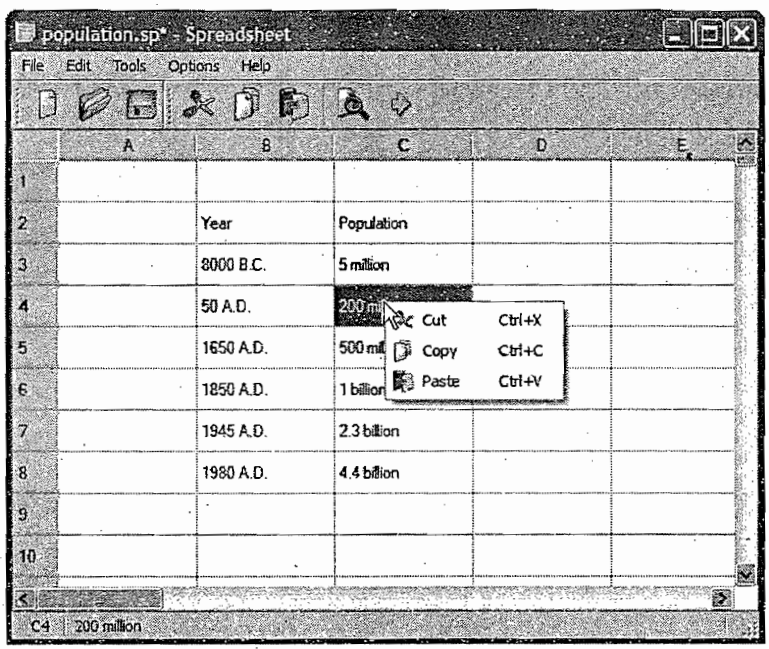


图 3.1　Spreadsheet 应用程序

在绝大多数图形用户界面应用程序的后台，都有一套提供底层功能的代码——例如，用于读写文件或者用于处理用户界面中的数据的代码。在第 4 章，将会再次把 Spreadsheet 应用程序当作实例，看看如何实现这些功能。

3.1　子类化 QMainWindow

通过子类化 QMainWindow，可以创建一个应用程序的主窗口。由于 QDialog 和 QMainWindow 都派生自 QWidget，所以在第 2 章中看到的许多创建对话框的技术，对于创建主窗口也同样适用。

可以使用 Qt 设计师创建应用程序的主窗口，但是在这一章，将使用代码来完成所有的功能，以便可以说明它们是如何完成的。如果你更喜欢可视化的方式，可以参考 Qt 设计师在线手册中的“Creating a Main Window Application”一章。

Spreadsheet 应用程序主窗口的源代码分别放在 mainwindow.h 和 mainwindow.cpp 中。先从头文件开始分析：

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

protected:
    void closeEvent(QCloseEvent *event);
```

我们将 MainWindow 类定义为 QMainWindow 类的子类。由于类 MainWindow 提供了自己的信号和槽,所以它包含了 Q_OBJECT宏。

closeEvent()函数是 QWidget 类中的一个虚函数,当用户关闭窗口时,这个函数会被自动调用。类 MainWindow 中重新实现了它,这样就可以向用户询问一个标准问题"Do you want to save your changes?",并且可以把用户的一些偏好设置保存到磁盘中。

```
private slots:
    void newFile();
    void open();
    bool save();
    bool saveAs();
    void find();
    void goToCell();
    void sort();
    void about();
```

像 File→New 和 Help→About 这样的菜单项,在 MainWindow 中会被实现为私有槽。除了 save()槽和 saveAs()槽返回一个 bool 值以外,绝大多数的槽都把 void 作为它们的返回值。当槽作为一个信号的响应函数而被执行时,就会忽略这个返回值;但是当把槽作为函数来调用时,其返回值对我们的作用就和调用任何一个普通的 C++ 函数时的作用是相同的。

```
    void openRecentFile();
    void updateStatusBar();
    void spreadsheetModified();

private:
    void createActions();
    void createMenus();
    void createContextMenu();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool okToContinue();
    bool loadFile(const QString &fileName);
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    void updateRecentFileActions();
    QString strippedName(const QString &fullFileName);
```

为了能够对用户界面提供支持,主窗口需要更多的私有槽以及数个私有函数。

```
    Spreadsheet *spreadsheet;
    FindDialog *findDialog;
    QLabel *locationLabel;
    QLabel *formulaLabel;
    QStringList recentFiles;
    QString curFile;

    enum { MaxRecentFiles = 5 };
    QAction *recentFileActions[MaxRecentFiles];
    QAction *separatorAction;

    QMenu *fileMenu;
    QMenu *editMenu;
    ...
    QToolBar *fileToolBar;
    QToolBar *editToolBar;
    QAction *newAction;
    QAction *openAction;
    ...
    QAction *aboutQtAction;
};

#endif
```

除了它自己的私有槽和私有函数以外，MainWindow 类还有很多私有变量。当用到这些私有槽和私有函数时，将再对它们进行解释。

现在来看看实现文件：

```
#include <QtGui>

#include "finddialog.h"
#include "gotocelldialog.h"
#include "mainwindow.h"
#include "sortdialog.h"
#include "spreadsheet.h"
```

我们包含了 <QtGui> 头文件，其中包含了在子类中所要用到的所有 Qt 类的定义。我们也包含了一些自定义头文件，特别是来自第 2 章的 finddialog.h、gotocelldialog.h 和 sortdialog.h 三个头文件。

```
MainWindow::MainWindow()
{
    spreadsheet = new Spreadsheet;
    setCentralWidget(spreadsheet);

    createActions();
    createMenus();
    createContextMenu();
    createToolBars();
    createStatusBar();

    readSettings();

    findDialog = 0;

    setWindowIcon(QIcon(":/images/icon.png"));
    setCurrentFile("");
}
```

在这个构造函数中，先从创建一个 Spreadsheet 窗口部件并且把它设置为这个主窗口的中央窗口部件开始。中央窗口部件会占用主窗口的中央区域部分（如图 3.2 所示）。Spreadsheet 类是 QTableWidget 类的一个子类，并且也具有一些电子制表软件的功能，如对电子制表软件公式的支持等。将会在第 4 章中实现这一功能。

私有函数 createActions()、createMenus()、createContextMenu()、createToolBars()和 createStatusBar()创建主窗口中的其余部分，readSettings()则读取这个应用程序存储的一些设置。

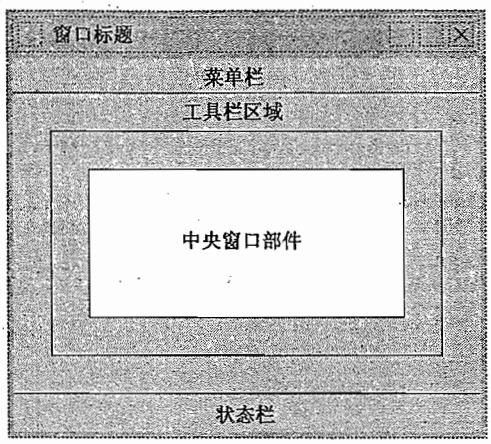


图 3.2 QMainWindow 中的区域分配

我们把 findDialog 指针初始化为空(null)指针。在第一次调用 MainWindow::find()函数时,将会创建该 FindDialog 对象。

在构造函数的最后部分,把窗口的图标设置为 icon.png,它是一个 PNG 格式的文件。Qt 支持很多图像格式,包括 BMP、GIF、JPEG、PNG、PNM、SVG、TIFF、XBM 和 XPM。调用 QWidget::setWindowIcon()函数可以设置显示在窗口左上角的图标。遗憾的是,还没有一种与平台无关的可在桌面上显示应用程序图标的设置方法。与平台相关的桌面图标设置方法在 http://doc.trolltech.com/4.3/appicon.html 中进行了阐述。

图形用户界面(GUI)应用程序通常会使用很多图片。为应用程序提供图片的方法有多种,如下是一些最常用的方法:

- 把图片保存到文件中,并且在运行时载入它们。
- 把 XPM 文件包含在源代码中。(这一方法之所以可行,是因为 XPM 文件也是有效的 C++ 文件。)
- 使用 Qt 的资源机制(resource mechanism)。

这里,使用了 Qt 的资源机制法,因为它比运行时载入文件的方法更方便,并且该方法适用于所支持的任意文件格式。我们将选中的图片存放在源代码树中名为 images 的子目录下。

为了利用 Qt 的资源系统(resource system),必须创建一个资源文件(resource file),并且在识别该资源文件的 .pro 文件中添加一行代码。在这个例子中,已经将资源文件命名为 spreadsheet.qrc,因此只需在 .pro 文件中添加如下一行代码:

```
RESOURCES = spreadsheet.qrc
```

资源文件自身使用了一种简单的 XML 文件格式。这里给出的是从已经使用的资源文件中摘录的部分内容:

```
<RCC>
<qresource>
    <file>images/icon.png</file>
    ...
    <file>images/gotocell.png</file>
</qresource>
</RCC>
```

所有资源文件都会被编译到应用程序的可执行文件中，因此并不会弄丢它们。当引用这些资源时，需要使用带路径前缀 :/(冒号斜线)的形式，这就是为什么会将图标表示成 :/images/icon.png 的形式。资源可以是任意类型的文件(并非只是一些图像)，并且可以在 Qt 需要文件名的大多数地方使用它们。第 12 章将对此做进一步说明。

3.2　创建菜单和工具栏

绝大多数现代图形用户界面应用程序都会提供一些菜单、上下文菜单和工具栏。菜单可以让用户浏览应用程序并且可以学会如何处理一些新的事情，上下文菜单和工具栏则提供了对那些经常使用的功能进行快速访问的方法。图 3.3 展示的是 Spreadsheet 应用程序的菜单。

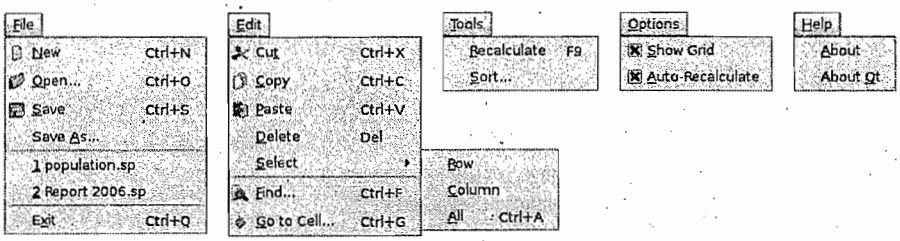


图 3.3　Spreadsheet 应用程序中的菜单

Qt 通过“动作”的概念简化了有关菜单和工具栏的编程。一个动作(action)就是一个可以添加到任意数量的菜单和工具栏上的项。在 Qt 中，创建菜单和工具栏包括以下这些步骤：

- 创建并且设置动作。
- 创建菜单并且把动作添加到菜单上。
- 创建工具栏并且把动作添加到工具栏上。

在这个 Spreadsheet 应用程序中，动作是在 createActions()函数中创建的：

```
void MainWindow::createActions()
{
    newAction = new QAction(tr("&New"), this);
    newAction->setIcon(QIcon(":/images/new.png"));
    newAction->setShortcut(QKeySequence::New);
    newAction->setStatusTip(tr("Create a new spreadsheet file"));
    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
```

动作 New 有一个加速键(New)、一个父对象(主窗口)、一个图标、一个快捷键和一个状态提示。大多数窗口系统都有用于特定动作的标准化的键盘快捷键。例如，在 Windows、KDE 和 GNOME 中，这个 New 动作就有一个快捷键 Ctrl + N，而在 Mac OS X 中则是 Command + N。通过使用适当的 QKeySequence::StandardKey 枚举值，就可以确保 Qt 能够为应用程序在其运行的平台上提供正确的快捷键。

把这个动作的 triggered()信号连接到主窗口的私有槽 newFile()——将会在下一节实现它。这个连接可以确保在用户选择 File→New 菜单项、选择工具栏上的 New 按钮或者按下 Ctrl + N 时，都可以调用 newFile()槽。

由于菜单中的 Open、Save 和 Save As 动作与 New 动作非常相似，所以将会直接跳到 File 菜单中的“recently opened files”(最近打开的文件)的部分。

```
    ...
    for (int i = 0; i < MaxRecentFiles; ++i) {
        recentFileActions[i] = new QAction(this);
        recentFileActions[i]->setVisible(false);
        connect(recentFileActions[i], SIGNAL(triggered()),
                this, SLOT(openRecentFile()));
    }
```

我们为 recentFileActions 数组添加动作。每个动作都是隐式的,并且会被连接到 openRecentFile()槽。稍后,将会看到如何让这些最新文件中的动作变得可见并且可用。

```
    exitAction = new QAction(tr("E&xit"), this);
    exitAction->setShortcut(tr("Ctrl+Q"));
    exitAction->setStatusTip(tr("Exit the application"));
    connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
```

这个 Exit 动作与目前为止所看到的那些动作稍微有些不同。由于没有用于终止应用程序的标准化键序列,所以需要在这里明确指定键序列。另外一个不同之处是:我们连接的是窗口的 close()槽,而它是由 Qt 提供的。

现在,可以跳到 Select All 动作中:

```
    ...
    selectAllAction = new QAction(tr("&All"), this);
    selectAllAction->setShortcut(QKeySequence::SelectAll);
    selectAllAction->setStatusTip(tr("Select all the cells in the "
                                     "spreadsheet"));
    connect(selectAllAction, SIGNAL(triggered()),
            spreadsheet, SLOT(selectAll()));
```

由于槽 selectAll()是由 QTableWidget 的父类之一的 QAbstractItemView 提供的,所以就没有必要再去亲自实现它。

现在,不妨进一步跳到 Options 菜单中的 Show Grid 动作中去:

```
    ...
    showGridAction = new QAction(tr("&Show Grid"), this);
    showGridAction->setCheckable(true);
    showGridAction->setChecked(spreadsheet->showGrid());
    showGridAction->setStatusTip(tr("Show or hide the spreadsheet's "
                                    "grid"));
    connect(showGridAction, SIGNAL(toggled(bool)),
            spreadsheet, SLOT(setShowGrid(bool)));
```

Show Grid 是一个复选(checkable)动作。复选动作在菜单中显示时会带一个复选标记,并且在工具栏中它可以实现成切换(toggle)按钮。当启用这个动作时,Spreadsheet 组件就会显示一个网格。我们用 Spreadsheet 组件的默认值来初始化这个动作,这样它们就可以从一开始就同步起来。然后,把 Show Grid 动作的 toggled(bool)信号和 Spreadsheet 组件的 setShowGrid(bool)槽连接起来,这个槽继承自 QTableWidget。一旦把这个动作添加到菜单或者工具栏中,用户就可以对网格的显示与否进行切换了。

Show Grid 动作和 Auto-Recalculate 动作是相互独立的两个复选动作。通过 QActionGroup 类的支持,Qt 也可以支持相互排斥的动作。

```
        ...
        aboutQtAction = new QAction(tr("About &Qt"), this);
        aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
        connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
    }
```

对于 About Qt 动作,通过访问 qApp 全局变量,我们可以使用 QApplication 对象的 aboutQt()槽。这个动作会弹出一个如图 3.4 所示的对话框。

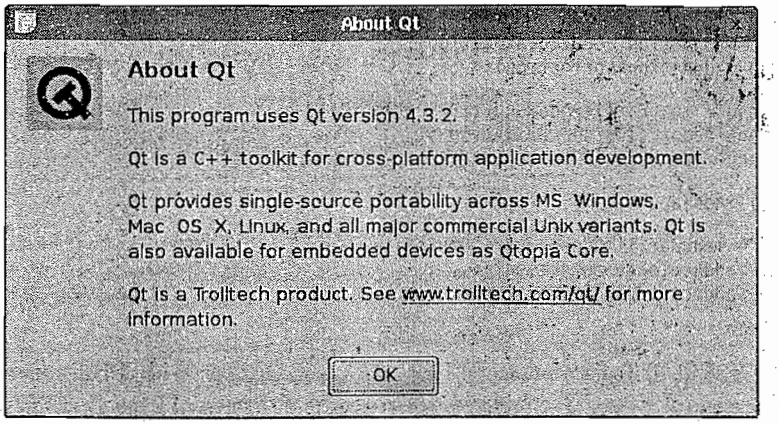


图 3.4 About Qt 对话框

现在已经创建了这些动作,还可以继续构建一个包含这些动作的菜单系统:

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(saveAsAction);
    separatorAction = fileMenu->addSeparator();
    for (int i = 0; i < MaxRecentFiles; ++i)
        fileMenu->addAction(recentFileActions[i]);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAction);
```

在 Qt 中,菜单都是 QMenu 的实例。addMenu()函数可以用给定的文本创建一个 QMenu 窗口部件,并且会把它添加到菜单栏中。QMainWindow::menuBar()函数返回一个指向 QMenuBar 的指针。菜单栏会在第一次调用 menuBar()函数的时候就创建出来。

从创建 File 菜单开始,然后再把 New、Open、Save 和 Save As 动作添加进去。插入一个间隔器(separator),可以从视觉上把关系密切的这些项放在一起。使用一个 for 循环从 recentFileActions 数组中添加一些动作(最初是隐藏起来的),然后在最后添加一个 exitAction 动作。

我们已经让一个指针指向了这些间隔器中的某一个。这样就可以允许隐藏(如果没有最近文件的话)或者显示那个间隔器,因为不希望出现在两个间隔器之间什么都没有的情况。

```
    editMenu = menuBar()->addMenu(tr("&Edit"));
    editMenu->addAction(cutAction);
    editMenu->addAction(copyAction);
    editMenu->addAction(pasteAction);
    editMenu->addAction(deleteAction);

    selectSubMenu = editMenu->addMenu(tr("&Select"));
    selectSubMenu->addAction(selectRowAction);
    selectSubMenu->addAction(selectColumnAction);
    selectSubMenu->addAction(selectAllAction);

    editMenu->addSeparator();
    editMenu->addAction(findAction);
    editMenu->addAction(goToCellAction);
```

现在来创建 Edit 菜单,就像在 File 菜单中所做的那样使用 QMenu::addMenu()函数添加各个动作,并且在希望出现子菜单的地方使用 QMenu::addMenu()函数添加子菜单。一个子菜单与它所属的菜单一样,也是一个 QMenu。

```
    toolsMenu = menuBar()->addMenu(tr("&Tools"));
    toolsMenu->addAction(recalculateAction);
    toolsMenu->addAction(sortAction);

    optionsMenu = menuBar()->addMenu(tr("&Options"));
    optionsMenu->addAction(showGridAction);
    optionsMenu->addAction(autoRecalcAction);

    menuBar()->addSeparator();

    helpMenu = menuBar()->addMenu(tr("&Help"));
    helpMenu->addAction(aboutAction);
    helpMenu->addAction(aboutQtAction);
}
```

通过类似的方式创建 Tools、Options 和 Help 菜单。在 Options 菜单和 Help 菜单之间插入一个间隔器。对于 Motif 和 CDE 风格,这个间隔器会把 Help 菜单放到菜单栏的最右端;对于其他的风格,则将会忽略这个间隔器。图 3.5 是这两种情况的示意图。

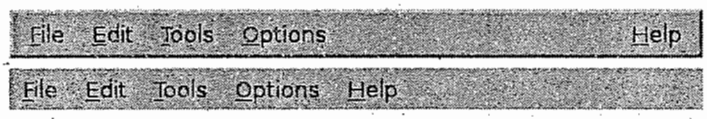


图 3.5 Motif 和 Windows 风格下的菜单栏

```
void MainWindow::createContextMenu()
{
    spreadsheet->addAction(cutAction);
    spreadsheet->addAction(copyAction);
    spreadsheet->addAction(pasteAction);
    spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

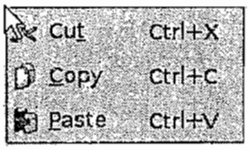


图 3.6 Spreadsheet 应用程序中的上下文菜单

任何 Qt 窗口部件都可以有一个与之相关联的 QActions 列表。要为该应用程序提供一个上下文菜单,可以将所需要的动作添加到 Spreadsheet 窗口部件中,并且将那个窗口部件的上下文菜单策略(context menu policy)设置为一个显示这些动作的上下文菜单。当用户在一个窗口部件上单击鼠标右键,或者是在键盘上按下一个与平台相关的按键时,就可以激活这些上下文菜单。Spreadsheet 中的上下文菜单如图 3.6 所示。

一种更为高级的提供上下文菜单方法是重新实现 QWidget::contextMenuEvent()函数,创建一个 QMenu 窗口部件,在其中添加所期望的那些动作,并且再对该窗口部件调用 exec()函数。

```
void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("&File"));
    fileToolBar->addAction(newAction);
    fileToolBar->addAction(openAction);
    fileToolBar->addAction(saveAction);

    editToolBar = addToolBar(tr("&Edit"));
    editToolBar->addAction(cutAction);
    editToolBar->addAction(copyAction);
    editToolBar->addAction(pasteAction);
    editToolBar->addSeparator();
    editToolBar->addAction(findAction);
    editToolBar->addAction(goToCellAction);
}
```

创建工具栏与创建菜单的过程很相似，我们据此创建一个 File 工具栏和一个 Edit 工具栏。就像菜单一样，工具栏也可以有多个间隔器，如图 3.7 所示。

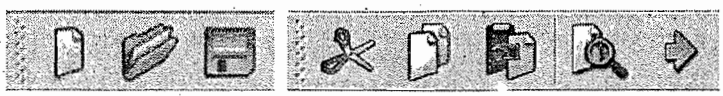


图 3.7　Spreadsheet 应用程序的工具栏

3.3　设置状态栏

随着菜单和工具栏的完成，已经为设置 Spreadsheet 应用程序的状态栏做好了准备。在程序的普通模式下，状态栏包括两个状态指示器：当前单元格的位置和当前单元格中的公式。状态栏也用于显示状态提示和其他一些临时消息。图 3.8 给出了各种情况下的状态栏。

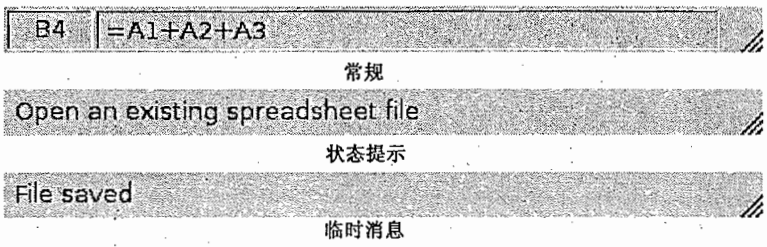


图 3.8　Spreadsheet 应用程序的状态栏

MainWindow 的构造函数会调用 createStatusBar()来设置状态栏。

```
void MainWindow::createStatusBar()
{
    locationLabel = new QLabel(" W999 ");
    locationLabel->setAlignment(Qt::AlignHCenter);
    locationLabel->setMinimumSize(locationLabel->sizeHint());

    formulaLabel = new QLabel;
    formulaLabel->setIndent(3);

    statusBar()->addWidget(locationLabel);
    statusBar()->addWidget(formulaLabel, 1);

    connect(spreadsheet, SIGNAL(currentCellChanged(int, int, int, int)),
            this, SLOT(updateStatusBar()));
    connect(spreadsheet, SIGNAL(modified()),
            this, SLOT(spreadsheetModified()));

    updateStatusBar();
}
```

QMainWindow::statusBar()函数返回一个指向状态栏的指针。[在第一次调用 statusBar()函数的时候会创建状态栏。]状态栏指示器是一些简单的 QLabel，可以在任何需要的时候改变它们的文本。已经在 formulaLabel 中添加了一个缩进格式，以便让那些在它里面显示的文本能够与它的左侧边有一个小的偏移量。当把这些 QLabel 添加到状态栏的时候，它们会自动被重定义父对象，以便让它们成为状态栏的子对象。

图 3.8 所示的两个标签都有不同的空间需求。单元格定位指示器只需要非常小的空间，并且在重新定义窗口大小时，任何多余的空间都会分配给位于右侧的单元格公式指示器。这是通过在公式标签的 QStatusBar::addWidget()调用中指定一个伸展因子 1 而实现的。位置指示器的默认伸展因子为 0，这也就意味着它不喜欢被伸展。

当 QStatusBar 摆放这些指示器窗口部件时,它会尽量考虑由 QWidget::sizeHint()提供的每一个窗口部件的理想大小,然后再对那些可伸展的任意窗口部件进行伸展以填满全部可用空间。一个窗口部件的理想大小取决于这个窗口部件的内容以及改变内容时的变化大小。为了避免对定位指示器连续不断地重定义大小,设置它的最小尺寸大小为它所能包含的最大字符数("W999")和一些空格的总大小。还把它的对齐方式设置为 Qt::AlignHCenter,以便可以在水平方向上居中对齐它的文本。

在函数结尾的附近,把 Spreadsheet 的两个信号和 MainWindow 的两个槽,updateStatusBar()和 spreadsheetModified(),连接了起来。

```
void MainWindow::updateStatusBar()
{
    locationLabel->setText(spreadsheet->currentLocation());
    formulaLabel->setText(spreadsheet->currentFormula());
}
```

updateStatusBar()槽可以更新单元格定位指示器和单元格公式指示器。只要用户把单元格光标移动到一个新的单元格,这个槽就会得到调用。该槽也可以作为一个普通函数而在 createStatusBar()的最后用于初始化这些指示器。因为 Spreadsheet 不会在一开始的时候就发射 currentCellChanged()消息,所以还必需这样做。

```
void MainWindow::spreadsheetModified()
{
    setWindowModified(true);
    updateStatusBar();
}
```

spreadsheetModified()槽把 windowModified 属性设置为 true,用以更新标题栏。这个函数也会更新位置和公式指示器,以便可以让它们反映事件的当前状态。

3.4 实现 File 菜单

在这一节中,将实现那些能够让 File 菜单项正常工作并且能够对最近打开文件进行管理的槽函数和私有函数。

```
void MainWindow::newFile()
{
    if (okToContinue()) {
        spreadsheet->clear();
        setCurrentFile("");
    }
}
```

当用户点击 File→New 菜单项或者单击工具栏上的 New 按钮时,就会调用 newFile()槽。如果存在还没有被保存的信息,okToContinue()私有函数就会弹出如图 3.9 所示的对话框:"Do you want to save your changes?"。如果用户选择 Yes 或者 No(保存文档应该选择 Yes),这个函数会返回 true;如果用户选择 Cancel,它就返回 false。Spreadsheet::clear()函数会清空电子制表软件中的全部单元格和公式。setCurrentFile()私有函数会更新窗口的标题,以说明正在编辑的是一个没有标题的文档,它还会设置 curFile 私有变量并且更新最近打开文件的列表。

```
bool MainWindow::okToContinue()
{
    if (isWindowModified()) {
        int r = QMessageBox::warning(this, tr("Spreadsheet"),
                     tr("The document has been modified.\n"
                        "Do you want to save your changes?"),
                     QMessageBox::Yes | QMessageBox::No
```

```
                        | QMessageBox::Cancel);
        if (r == QMessageBox::Yes) {
            return save();
        } else if (r == QMessageBox::Cancel) {
            return false;
        }
    }
    return true;
}
```

在 okToContinue()函数中,会检测 windowModified 属性的状态。如果该属性的值是 true,就显示一个如图 3.9 所示的消息框。这个消息框包含一个 Yes 按钮、一个 No 按钮和一个 Cancel 按钮。

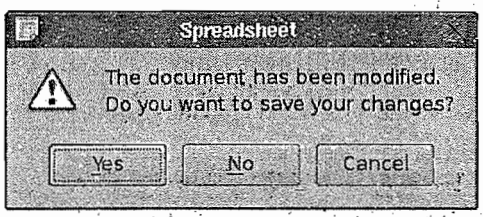


图 3.9 "Do you want to save your changes?"消息框

QMessageBox 提供了许多标准按钮,并且会自动尝试着让其中的一个成为默认的确认按钮(在用户按下 Enter 键时会得到激活),一个成为默认的退出按钮(在用户按下 Esc 时会得到激活)。选择一些特殊的按钮作为默认的确认按钮和退出按钮也是有可能的,用户还可以自定义按钮中将要显示的文本内容。

当首次看到 warning()函数调用时,可能会觉得它有点复杂,但这种常用语法实际上是相当简单的:

```
QMessageBox::warning(parent, title, message, buttons);
```

除了 warning()之外,QMessageBox 还提供了 information()、question()和 critical()函数,它们每一个都有自己特定的图标,这些图标如图 3.10 所示。

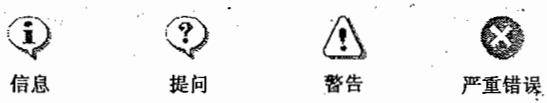


图 3.10 Windows 风格下的消息框图标

```
void MainWindow::open()
{
    if (okToContinue()) {
        QString fileName = QFileDialog::getOpenFileName(this,
                                tr("Open Spreadsheet"), ".",
                                tr("Spreadsheet files (*.sp)"));
        if (!fileName.isEmpty())
            loadFile(fileName);
    }
}
```

open()槽对 File→Open 做出响应。就像 newFile()一样,它首先调用 okToContinue()函数来处理任何没有被保存的变化。然后它使用方便的 QFileDialog::getOpenFileName()静态函数从用户那里获得一个新的文件名。这个函数会弹出一个文件对话框,让用户选择一个文件,并且返回这个文件名——或者,如果用户单击了 Cancel 按钮,则返回一个空字符串。

传递给 QFileDialog::getOpenFileName()函数的第一个参数是它的父窗口部件。用于对话框和其他窗口部件的这种父子对象关系意义并不相同。对话框通常都拥有自主权,但是如果它有父对

象,那么在默认情况下,它就会居中放到父对象上。一个子对话框也会共用它的父对象的任务栏。

第二个参数是这个对话框应当使用的标题。第三个参数告诉它应当从哪一级目录开始,在这个例子中就是当前目录。

第四个参数指定了文件过滤器。文件过滤器(filter)由一个描述文本和一个通配符组成。如果除了要支持 Spreadsheet 本地文件格式以外,还需要支持采用逗号分隔的数据文件和 Lotus 1-2-3 文件,就应当使用如下的文件过滤器:

```
tr("Spreadsheet files (*.sp)\n"
   "Comma-separated values files (*.csv)\n"
   "Lotus 1-2-3 files (*.wk1 *.wks)")
```

loadFile()私有函数是在 open()中得到调用的,它用来载入文件。我们让它成为一个独立的函数,是因为会在载入最近打开的文件中使用同样的功能。

```
bool MainWindow::loadFile(const QString &fileName)
{
    if (!spreadsheet->readFile(fileName)) {
        statusBar()->showMessage(tr("Loading canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"), 2000);
    return true;
}
```

我们使用 Spreadsheet::readFile()函数从磁盘中读取文件。如果载入成功,会调用 setCurrentFile()函数来更新这个窗口的标题;否则,Spreadsheet::readFile()将会通过一个消息框把遇到的问题通知给用户。在通常情况下,让底层组件来报告错误消息是一个不错的习惯,这是因为它们可以提供准确的错误细节信息。

在上述两种情况下,都会在状态栏中显示一个消息 2 秒(2000 毫秒),这样可以通知用户应用程序正在做什么。

```
bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

bool MainWindow::saveFile(const QString &fileName)
{
    if (!spreadsheet->writeFile(fileName)) {
        statusBar()->showMessage(tr("Saving canceled"), 2000);
        return false;
    }

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}
```

save()槽对 File→Save 做出响应。如果因为这个文件是之前打开的文件或者它是一个已经保存过的文件,这样已经有了一个名字,那么 save()函数就会用这个名字调用 saveFile()函数;否则,它只是简单地调用 saveAs()函数。

```
bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
```

```
                                tr("Save Spreadsheet"), ".",
                                tr("Spreadsheet files (*.sp)"));
    if (fileName.isEmpty())
        return false;

    return saveFile(fileName);
}
```

saveAs()槽对 File→Save As 做出响应。调用 QFileDialog::getSaveFileName()函数来从用户那里得到一个文件名。如果用户单击了 Cancel,则返回 false,这将会使这个结果向上传递给它的调用者[save()或者 okToContinue()]。

如果给定的文件已经存在,getSaveFileName()函数将会要求用户确认是否需要覆盖该文件。但通过给 getSaveFileName()函数传递一个 QFileDialog::DontConfirmOverwrite 附加参数,则可以改变这一行为。

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

当用户单击 File→Exit 或者单击窗口标题栏中的关闭按钮时,将会调用 QWidget::close()槽。该槽会给这个窗口部件发射一个"close"事件。通过重新实现 QWidget::closeEvent()函数,就可以中途截取对这个主窗口的关闭操作,并且可以确定到底是不是真的要关闭这个窗口。

如果存在未保存的更改并且用户选择了 Cancel,就会"忽略"这个关闭事件并且让这个窗口不受该操作的影响。一般情况下,我们会接受这个事件,这会让 Qt 隐藏该窗口。也可以调用私有函数 writeSettings()来保存这个应用程序的当前设置。

当最后一个窗口关闭后,这个应用程序就结束了。如果需要,通过把 QApplication 的 quitOnLastWindowClosed 属性设置为 false,可以禁用这种行为。在这种情况下,该应用程序将会持续保持运行,直到调用 QApplication::quit()函数,程序才会结束。

```
void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setWindowModified(false);

    QString shownName = tr("Untitled");
    if (!curFile.isEmpty()) {
        shownName = strippedName(curFile);
        recentFiles.removeAll(curFile);
        recentFiles.prepend(curFile);
        updateRecentFileActions();
    }

    setWindowTitle(tr("%1[*] - %2").arg(shownName)
                                   .arg(tr("Spreadsheet")));
}

QString MainWindow::strippedName(const QString &fullFileName)
{
    return QFileInfo(fullFileName).fileName();
}
```

在 setCurrentFile()中,对保存正在编辑的文件名的 curFile 私有变量进行了设置。在把这个文件名显示在标题栏中之前,需要使用 strippedName()函数移除文件名中的路径字符,这样可以使文件名看起来更友好一些。

每个 QWidget 都有一个 windowModified 属性,如果该窗口的文档存在没有保存的变化,则应当把它设置为 true,否则应当将其设置为 false。在 Mac OS X 下,未保存的文档是通过窗口标题栏上关闭按钮中的一个点来表示的;在其他平台下,则是通过文件名字后跟一个星号来表示的。Qt 会自动处理这一行为,只要始终让 windowModified 属性保持为当前最新状态,并且当需要显示星号的时候,把"[*]"标记放在窗口的标题栏上即可。

传递给 setWindowTitle()函数的文本是:

```
tr("%1[*] - %2").arg(shownName)
                .arg(tr("Spreadsheet"))
```

QString::arg()函数将会使用自己的参数替换最小数字的"%n"参数,并且会用它的参数返回结果"%n"字符和最终的结果字符串。在本例中,arg()被用于两个"%n"参数中。第一个 arg()调用会替换参数"%1",第二个 arg()调用则会替换参数"%2"。如果文件名是 budget.sp 并且没有载入翻译文件,那么结果字符串将是"budget.sp[*] - Spreadsheet"。这本应更简单地写作如下代码:

```
setWindowTitle(shownName + tr("[*] - Spreadsheet"));
```

但使用 arg()函数可以为翻译人员提供更多的灵活性。

如果存在文件名,就需要更新应用程序的最近打开文件列表 recentFiles。可以调用 removeAll()从列表中移除任何已经出现过的文件名,从而避免该文件名的重复。然后,可以调用 prepend()把这个文件名作为文件列表的第一项添加进去。在更新了文件列表之后,可以调用私有函数updateRecentFileActions()更新 File 菜单中的那些条目。

```
void MainWindow::updateRecentFileActions()
{
    QMutableStringListIterator i(recentFiles);

    while (i.hasNext()) {
        if (!QFile::exists(i.next()))
            i.remove();
    }

    for (int j = 0; j < MaxRecentFiles; ++j) {
        if (j < recentFiles.count()) {
            QString text = tr("&%1 %2")
                           .arg(j + 1)
                           .arg(strippedName(recentFiles[j]));
            recentFileActions[j]->setText(text);
            recentFileActions[j]->setData(recentFiles[j]);
            recentFileActions[j]->setVisible(true);
        } else {
            recentFileActions[j]->setVisible(false);
        }
    }
    separatorAction->setVisible(!recentFiles.isEmpty());
}
```

使用一个 Java 风格的迭代器,可以移除任何不再存在的文件。一些文件或许已经在前面的会话中使用过,但在此之前还没被删除掉。recentFiles 变量的类型是 QStringList(QString 型列表)。第 11 章会详细说明一些像 QStringList 一样的容器类,其中将会说明它们与 C++ 标准模板库(Standard Template Library,STL)之间的关系,也会说明 Qt 的 Java 风格迭代器类的用法。

然后,再遍历一次文件列表,这一次使用数组风格的索引形式。对于每一个文件,创建一个由一个与操作符、一位数字(j+1)、一个空格和该文件名(不带路径)组成的字符串。我们要为使用这种文本设置相应的动作。例如,如果第一个文件是 C:\My Documents\tab04.sp,那么第一个动作的文本将会是"&1 tab04.sp"。图 3.11 给出了 recentFileActions 数组和菜单的最终结果之间的对应关系。

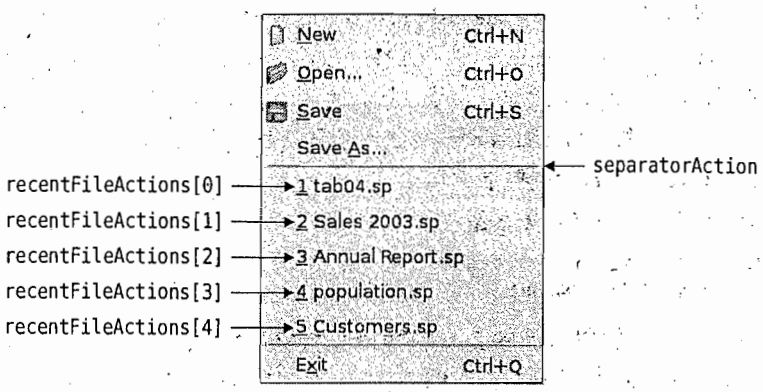


图 3.11 带最近打开文件列表的 File 菜单

每一个动作都可以带一个与之相关的 QVariant 型 data 项。QVariant 类型可以保存许多 C++ 和 Qt 型变量,第 11 章将说明这一点。这里,将文件的全名保存在动作的 data 项中,以便随后可以方便地找到它。还要将这个动作设置为可见。

如果有比最新文件更多的文件动作,那么只需隐藏那些多余的动作即可。最后,如果至少还存在一个最近打开的文件,那么就应该把间隔器设置为可见。

```
void MainWindow::openRecentFile()
{
    if (okToContinue()) {
        QAction *action = qobject_cast<QAction *>(sender());
        if (action)
            loadFile(action->data().toString());
    }
}
```

当用户选择了一个最近打开的文件,就会调用 openRecentFile()槽。只要有任何未保存的变化,就会调用 okToContinue()函数,并且假定用户没有取消,还可以使用 QObject::sender()查出是哪个特有动作调用了这个槽。

qobject_cast <T> () 函数可在 Qt 的 moc(meta-object compiler,元对象编译器)所生成的元信息基础上执行动态类型强制转换(dynamic cast)。它返回一个指向所需 QObject 子类的指针,或者是在该对象不能被转换成所需的那种类型时返回 0。与标准 C++ 的 dynamic_cast <T> () 不同,Qt 的 qobject_cast <T> () 可正确地跨越动态库边界。在例子中,使用 qobject_cast <T> () 把一个 QObject 指针转换成 QAction 指针。如果这个转换是成功的(应当是这样的),就可以利用从动作的 data 项中所提取的文件全名来调用 loadFile()函数。

顺便值得一提的是,由于知道这个发射器是一个 QAction,如果使用 static_cast <T> () 或者传统的 C 风格的数据类型强制转换代替原有的数据转换方式,这个程序应当仍然是可以运行的。请参见附录 D 中"类型转换"一节对不同 C++ 数据类型强制转换的概述。

3.5 使用对话框

这一节将说明如何在 Qt 中使用对话框——如何创建、初始化以及运行它们,并且对用户交互中的选择做出响应。本节将会使用在第 2 章中创建的 Find、Go to Cell 和 Sort 对话框,也会创建一个简单的 About 对话框。

我们从如图 3.12 所示的 Find 对话框开始。由于希望用户能够在 Spreadsheet 窗口和 Find 对话

框之间进行切换,所以 Find 对话框必须是非模态(modeless)的。非模态窗口就是运行在应用程序中对于任何其他窗口都独立的窗口。

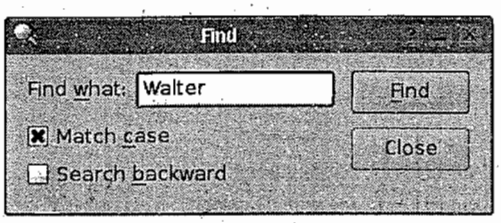


图 3.12 Spreadsheet 应用程序的 Find 对话框

创建非模态对话框时,通常会把它的信号连接到能够对用户的交互做出响应的那些槽上。

```
void MainWindow::find()
{
    if (!findDialog) {
        findDialog = new FindDialog(this);
        connect(findDialog, SIGNAL(findNext(const QString &,
                                            Qt::CaseSensitivity)),
                spreadsheet, SLOT(findNext(const QString &,
                                           Qt::CaseSensitivity)));
        connect(findDialog, SIGNAL(findPrevious(const QString &,
                                                Qt::CaseSensitivity)),
                spreadsheet, SLOT(findPrevious(const QString &,
                                               Qt::CaseSensitivity)));
    }

    findDialog->show();
    findDialog->raise();
    findDialog->activateWindow();
}
```

Find 对话框是一个可以让用户在电子制表软件中搜索文本的窗口。当用户单击 Edit→Find 时,就会调用 find()槽来弹出 Find 对话框。这时,就可能出现下列几种情形:

- 这是用户第一次调用 Find 对话框。
- 以前曾经调用过 Find 对话框,但用户关闭了它。
- 以前曾经调用过 Find 对话框,并且现在它还是可见的。

如果 Find 对话框还不曾存在过,就可以创建它并且把它的 findNext()信号和 findPrevious()信号与 Spreadsheet 中相对应的那些槽连接起来。本应该在 MainWindow 的构造函数中创建这个对话框,但是推迟对话框的创建过程将可以使程序的启动更加快速。还有,如果从来没有使用到这个对话框,那么它就决不会被创建,这样可以既节省时间又节省内存。

然后,调用 show()、raise()和 activateWindow()来确保窗口位于其他窗口之上并且是可见的和激活的。只调用 show()就足以让一个隐藏窗口变为可见的、位于最上方并且是激活的,但是也有可能是在 Find 对话框窗口已经是可见的时候又再次调用了它,在这种情况下,show()调用可能什么也不做,那么就必须调用 raise()和 activateWindow()让窗口成为顶层窗口和激活状态。还有另外一种方法,本可以写成:

```
if (findDialog->isHidden()) {
    findDialog->show();
} else {
    findDialog->raise();
    findDialog->activateWindow();
}
```

但这样的程序就好像明明在穿越单行道却又同时去看这条单行道的两个方向一样,显得多余。

现在看一下如图3.13所示的Go to Cell对话框。我们希望用户可以弹出、使用和关闭它,但是却不希望让这个窗口能够与应用程序中的其他窗口相互切换。也就是说,Go to Cell对话框窗口必须是模态(modal)的。模态窗口就是一个在得到调用可以弹出并可以阻塞应用程序的窗口,从而会从调用发生开始起妨碍其他的任意处理或者交互操作,直到关闭该窗口为止。前面使用的文件对话框和消息框就是模态的。

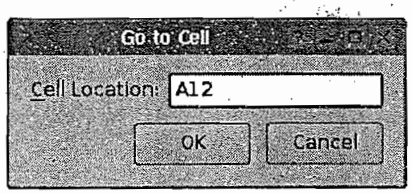


图3.13 Spreadsheet应用程序的Go to Cell对话框

如果对话框是通过show()调用的,那么它就是非模态对话框[除非此后又调用了setModal(),才会让它变为模态对话框]。但是,如果它是通过exec()调用的,那么该对话框就会是模态对话框。

```
void MainWindow::goToCell()
{
    GoToCellDialog dialog(this);
    if (dialog.exec()) {
        QString str = dialog.lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                    str[0].unicode() - 'A');
    }
}
```

如果对话框被接受,函数QDialog::exec()可返回一个true值(QDialog::Accepted),否则就会返回一个false值(QDialog::Rejected)。可以回想一下,当初在第2章利用Qt设计师创建Go to Cell对话框时,就曾经把OK连接到accept(),把Cancel连接到reject()。如果用户选择OK,就把当前单元格的值设置成行编辑器中的值。

QTable::setCurrentCell()函数需要两个参数:一个行索引和一个列索引。在Spreadsheet应用程序中,单元格A1就是单元格(0,0),单元格B27就是单元格(26,1)。为了从函数QLineEdit::text()返回的QString中获得行索引,可以使用QString::mid()来提取行号(这个函数将返回一个从字符串的开始直到末尾位置的子字符串),然后使用QString::toInt()把它转换成一个整数值,并且把该值再减去1。对于列号,则可以用这个字符串中第一个字符的大写数值减去字符'A'的数值而得到。我们知道,该字符串将具有正确的格式,因为为对话框创建了一个QRegExpValidator检验器,只有满足一个字符后面再跟至多三位数字格式的字符串才能让OK按钮起作用。

goToCell()函数与目前看到的所有代码都有些不同,因为它在堆栈中创建了一个作为变量的窗口部件(一个GoToCellDialog)。虽然多使用了一行代码,但是换来了使用new和delete的简便:

```
void MainWindow::goToCell()
{
    GoToCellDialog *dialog = new GoToCellDialog(this);
    if (dialog->exec()) {
        QString str = dialog->lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                    str[0].unicode() - 'A');
    }
    delete dialog;
}
```

由于在使用完一个对话框(或者菜单)后,通常就不再需要它了,所以在堆栈中创建对话框(和上下文菜单)是一种常见的编程模式,并且对话框会在作用域结束后自动销毁掉。

现在转到 Sort 对话框上。Sort 对话框是一个模态对话框,它允许用户在当前的选定区域中使用给定的列进行排序。图 3.14 给出了一个排序的实例,用列 B 作为排序的主键,列 A 作为排序的第二键(两个都采用升序)。

	A	B	C
1	George	Washington	1789-1797
2	John	Adams	1797-1801
3	Thomas	Jefferson	1801-1809
4	James	Madison	1809-1817
5	James	Monroe	1817-1825
6	John Quincy	Adams	1825-1829
7	Andrew	Jackson	1829-1837
8			

(a) 排序前

	A	B	C
1	John	Adams	1797-1801
2	John Quincy	Adams	1825-1829
3	Andrew	Jackson	1829-1837
4	Thomas	Jefferson	1801-1809
5	James	Madison	1809-1817
6	James	Monroe	1817-1825
7	George	Washington	1789-1797
8			

(b) 排序后

图 3.14 对电子制表软件的选定区域进行排序

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());

    if (dialog.exec()) {
        SpreadsheetCompare compare;
        compare.keys[0] =
              dialog.primaryColumnCombo->currentIndex();
        compare.keys[1] =
              dialog.secondaryColumnCombo->currentIndex() - 1;
        compare.keys[2] =
              dialog.tertiaryColumnCombo->currentIndex() - 1;
        compare.ascending[0] =
              (dialog.primaryOrderCombo->currentIndex() == 0);
        compare.ascending[1] =
              (dialog.secondaryOrderCombo->currentIndex() == 0);
        compare.ascending[2] =
              (dialog.tertiaryOrderCombo->currentIndex() == 0);
        spreadsheet->sort(compare);
    }
}
```

sort()函数中的代码使用了一种和 goToCell()函数中用到的类似模式:

- 在堆栈中创建对话框并且对其进行初始化。
- 使用 exec()弹出对话框。
- 如果用户单击 OK,就从对话框的各个窗口部件中提取并且使用这些用户输入的值。

setColumnRange()调用将那些可用于排序的列变量设置为选定的列。例如,使用图 3.14 中的选择,range.leftColumn()将返回值 0,即 'A' + 0 = 'A',并且 range.rightColumn()将返回值 2,即 'A' + 2 = 'C'。

compare 对象储存了主键、第二键和第三键以及它们的排序顺序。(将会在下一章中看到 SpreadsheetCompare 类的定义。)这个对象会由 Spreadsheet::sort()使用,用于两行的比较。keys 数组存储了这些键的列号。例如,如果选择区域是从 C2 扩展到 E5,那么列 C 的位置就是 0。ascending

数组中按 bool 格式存储了和每一个键相关的顺序。QComboBox::currentIndex()返回当前选定项的索引值,该值是一个从 0 开始的数。对于第二键和第三键,考虑到"None"项,我们从当前项减去 1。

sort()函数会完成这项工作,但是它显得稍有不足。它认为 Sort 对话框是按照一种特定的方式来实现的,也就是像上面那样来处理组合框和"None"项。这就意味着,如果重新设计了 Sort 对话框,也许就需要重新编写这些代码。如果对话框只会从一个地方调用,那么这样的方式应该是足够了,但是如果对话框可能会在几个地方调用到,那么这种处理方式就等于打开了维护工作的梦魇之门。

一种更为稳健的方法是让 SortDialog 类具有自适应性,这可以通过让它自己创建一个 SpreadsheetCompare 对象,然后使这个对象只能被它的调用者使用来做到这一点。这样就可以有效地简化 MainWindow::sort()函数:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());

    if (dialog.exec())
        spreadsheet->performSort(dialog.comparisonObject());
}
```

这种方法可产生松散的耦合组件,并且当从多个地方调用该对话框时,它几乎总可以做出正确的选择。

一种更为极端的方式是在初始化 SortDialog 对象的时候就为其传递一个指向 Spreadsheet 对象的指针,并且允许对话框直接对 Spreadsheet 进行操作。这样做会使 SortDialog 少一些通用性,因为它仅能适用于一种类型的窗口部件,但是通过去除 SortDialog::setColumnRange()函数,它的确是进一步简化了程序代码。于是,现在的 MainWindow::sort()函数将变成如下所示的样子:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    dialog.setSpreadsheet(spreadsheet);
    dialog.exec();
}
```

相比而言,第一种方法中调用者需要知道与这个对话框相关的暗示信息,而第二种方法中的对话框需要知道由调用者所提供的与数据结构相关的暗示信息。在对话框需要作用于现场变化的地方,这种方法显得更为有用些。但是就像第一种方法中调用者的代码功能不足一样,如果数据结构发生了变化,则第三种方法也会失效。

一些开发者只会选用一种对话框处理方法并对其持之以恒。这有一个好处,就是能够精通和简练处理方法,因为所有的对话框都使用的是同一种处理模式,但是这也会失去对调用对话框时没有用到的那些其他有益处理方法。理想情况下,应根据每一个对话框的自身来选择应当使用的对话框处理方法。

我们将用 About 对话框来圆满结束这一节。可以创建一个像 Find 或 Go to Cell 对话框那样的自定义对话框来显示应用程序的有关信息,但是因为绝大多数 About 对话框都具有较为固定的格式,所以 Qt 提供了一种更为简单的解决方案。

```
void MainWindow::about()
{
    QMessageBox::about(this, tr("About Spreadsheet"),
            tr("<h2>Spreadsheet 1.1</h2>"
               "<p>Copyright &copy; 2008 Software Inc."
```

```
                "<p>Spreadsheet is a small application that "
                "demonstrates QAction, QMainWindow, QMenuBar, "
                "QStatusBar, QTableWidget, QToolBar, and many other "
                "Qt classes."));
}
```

通过调用一个方便的静态函数 QMessageBox::about(),就可以获得 About 对话框。这个函数和 QMessageBox::warning()的形式非常相似,只是它使用了父窗口的图标,而不是标准的"警告"图标。这个对话框的最终结果显示在图 3.15 中。

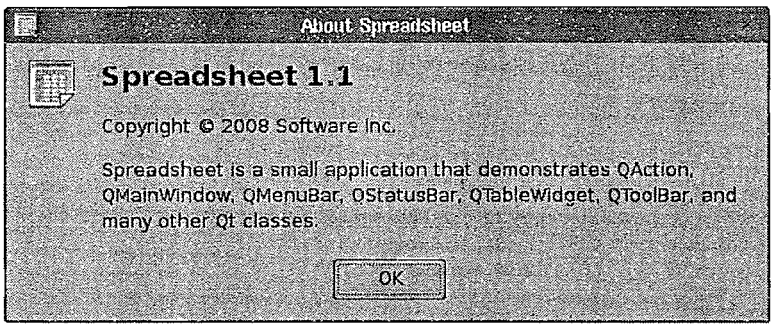


图 3.15 Spreadsheet 的 About 对话框

到现在为止,已经使用了由 QMessageBox 和 QFileDialog 提供的多个方便的静态函数。这些函数可以创建一个对话框,初始化它,并且可以对它调用 exec()。当然,也可以像创建其他任意窗口部件一样创建 QMessageBox 或者 QFileDialog 窗口部件,然后再明确地对它调用 exec(),或者甚至是 show(),尽管这样的处理方式会显得有些不大方便。

3.6 存储设置

在 MainWindow 的构造函数中,调用了 readSettings()来载入应用程序存储的那些设置。与之相似的是,在 closeEvent()中,调用 writeSettings()来保存这些设置。这两个函数是最后两个需要实现的 MainWindow 成员函数。

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    settings.setValue("geometry", saveGeometry());
    settings.setValue("recentFiles", recentFiles);
    settings.setValue("showGrid", showGridAction->isChecked());
    settings.setValue("autoRecalc", autoRecalcAction->isChecked());
}
```

writeSettings()函数保存了主窗口的几何形状(位置和尺寸大小)、最近打开文件列表以及 Show Grid 和 Auto-Recalculate 选项的设置值。

默认情况下,QSettings 会存储应用程序中与特定平台相关的一些设置信息。在 Windows 系统中,它使用的是系统注册表;在 UNIX 系统中,它会把设置信息存储在文本文件中;在 Mac OS X 中,它会使用 Core Foundation Preferences 的应用程序编程接口。

构造函数的参数说明了组织的名字和应用程序的名字。采用与平台相关的方式,可以利用这一信息查找这些设置所在的位置。

QSettings 把设置信息存储为键值对(key-value pair)的形式。键(key)与文件系统的路径很相

似。可以使用路径形式的语法(例如,findDialog/matchCase)来指定子键(subkey)的值,或者也可以使用 beginGroup()和 endGroup()的形式:

```
settings.beginGroup("findDialog");
settings.setValue("matchCase", caseCheckBox->isChecked());
settings.setValue("searchBackward", backwardCheckBox->isChecked());
settings.endGroup();
```

值(value)可以是一个 int、bool、double、QString、QStringList 或者是 QVariant 所支持的其他任意类型,包括那些已经注册过的自定义类型。

```
void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    restoreGeometry(settings.value("geometry").toByteArray());

    recentFiles = settings.value("recentFiles").toStringList();
    updateRecentFileActions();

    bool showGrid = settings.value("showGrid", true).toBool();
    showGridAction->setChecked(showGrid);

    bool autoRecalc = settings.value("autoRecalc", true).toBool();
    autoRecalcAction->setChecked(autoRecalc);
}
```

readSettings()函数可以载入之前使用 writeSettings()函数所保存的那些设置。value()函数中的第二个参数可以在没有可用设置的情况下指定所需的默认值。在应用程序第一次运行时,使用的就是这些默认值。由于没有为形状或者最近打开文件列表指定第二个参数,所以在第一次运行时,窗口会使用任意但是却合理的大小和位置,而最近文件列表会是一个空表。

在 readSettings()和 writeSettings()中使用与 QSettings 相关的全部代码为 MainWindow 所选择的布置方案,都只是许多可用方案中的一种而已。可以在应用程序执行期间的任何时候和程序代码中的任何地方,随时随地创建一个 QSettings 对象,用它查询或者修改一些设置。

现在已经完成了对 Spreadsheet 的 MainWindow 的实现。在后续的几节中,将会讨论如何修改 Spreadsheet 应用程序来让它可以处理多文档以及如何实现一个程序启动画面(splash screen)。将会在下一章中完成它的功能,包括公式和排序的处理。

3.7 多文档

现在,已经为编写 Spreadsheet 应用程序的 main()函数的代码做好了准备:

```
#include <QApplication>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow mainWin;
    mainWin.show();
    return app.exec();
}
```

这个 main()函数和以前曾经写过的那些函数稍微有点不同:以变量的形式在堆栈里创建 MainWindow 实例,而不是使用 new 来创建它。当函数结束时,MainWindow 实例会自动销毁。

就像上面 main()函数所显示的那样,Spreadsheet 应用程序只提供了一个单一主窗口,并且在同一时间只能处理一个文档。如果想让它在同一时间具有处理多个文档的能力,就需要同时启动多

个 Spreadsheet 应用程序实例。但是这对于用户来讲是很不方便的,用户需要的是一个可以处理多个文档的单一应用程序实例,就像在一个网页浏览器实例中可以同时提供多个浏览器窗口一样。

下面将修改 Spreadsheet 应用程序,以使它可以处理多个文档。首先,需要对 File 菜单做一些简单改动:

- 利用 File→New 创建一个空文档主窗口,而不是再次使用已经存在的主窗口。
- 利用 File→Close 关闭当前主窗口。
- 利用 File→Exit 关闭所有窗口。

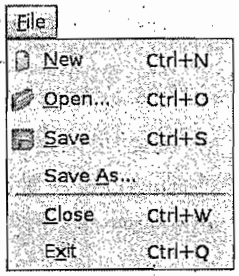


图 3.16 新的 File 菜单

在 File 菜单的最初版本中,并没有 Close 选项,这只是因为当时它还和 Exit 一样具有相同的功能。新的 File 菜单如图 3.16 所示。

新的 main()函数为:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
    return app.exec();
}
```

具有多窗口功能后,现在就需要使用菜单中的 new 来创建 MainWindow。考虑到节省内存,可以在工作完成之后使用 delete 操作删除主窗口。

这是新的 MainWindow::newFile()槽:

```
void MainWindow::newFile()
{
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
}
```

我们只创建了一个新的 MainWindow 实例。这看起来有些奇怪,因为没有保留指向这个新窗口的任何指针,但实际上这并不是什么问题,因为 Qt 会对所有的窗口进行跟踪。

以下是用于 Close 和 Exit 的动作:

```
void MainWindow::createActions()
{
    ...
    closeAction = new QAction(tr("&Close"), this);
    closeAction->setShortcut(QKeySequence::Close);
    closeAction->setStatusTip(tr("Close this window"));
    connect(closeAction, SIGNAL(triggered()), this, SLOT(close()));

    exitAction = new QAction(tr("E&xit"), this);
    exitAction->setShortcut(tr("Ctrl+Q"));
    exitAction->setStatusTip(tr("Exit the application"));
    connect(exitAction, SIGNAL(triggered()),
            qApp, SLOT(closeAllWindows()));
    ...
}
```

槽 QApplication::closeAllWindows()会关闭所有应用程序的窗口,除非其中一个应用程序拒绝了这个关闭事件。这正是在此所需的行为。我们不用再考虑那些有关是否保存的事情,因为就算关闭一个窗口,都会在 MainWindow::closeEvent()中处理这些情况。

看起来好像已经完成了应用程序对多窗口处理能力的工作。遗憾的是,这里还隐藏着一个潜在的问题:如果用户一直创建并且关闭主窗口,那么这台机器早晚会耗尽它的全部内存。这是因为我们

保存了 newFile()中创建的 MainWindow 窗口部件,但是从没有删除它们。当用户关闭一个主窗口时,默认行为是隐藏它,所以它还会保留在内存中。对于如此多的主窗口,的确会造成一定的问题。

解决办法是在构造函数中对 Qt::WA_DeleteOnClose 的属性进行设置:

```
MainWindow::MainWindow()
{
    ...
    setAttribute(Qt::WA_DeleteOnClose);
    ...
}
```

这样做就会告诉 Qt 在关闭窗口时将其删除。Qt::WA_DeleteOnClose 属性是可以在 QWidget 上进行设置并用来影响这个窗口部件的行为的诸多标记之一。

内存泄漏并不是必须处理的唯一问题。最初的应用程序设计包含了一个隐含的假设,也就是它仅有一个主窗口。对于多窗口,每一个主窗口都有它自己的最近打开文件列表和它自己的一些选项。很明显,最近打开文件列表对于整个应用程序来说应该是全局的。通过把 recentFiles 变量声明为静态变量,可以相当容易地解决这个问题,这样对于整个应用程序来说,会只存在一个该列表的实例。但随后必须确保的是:无论何时调用 updateRecentFileActions()函数来更新 File 菜单,都必须是在所有的主窗口上调用它。这是实现这一做法的代码:

```
foreach (QWidget *win, QApplication::topLevelWidgets()) {
    if (MainWindow *mainWin = qobject_cast<MainWindow *>(win))
        mainWin->updateRecentFileActions();
}
```

这段代码使用了 Qt 的 foreach 结构体(将在第 11 章中对其进行说明)来遍历这个应用程序的所有窗口,并且对所有类型为 MainWindow 的窗口部件调用 updateRecentFileActions()。可以使用类似的代码来同步 Show Grid 和 Auto-Recalculate 选项,或者用于确保同一个文件不会被加载两次。

在每一个主窗口中只提供一个文档的应用程序称为单文档界面(single document interface,SDI)应用程序。在 Windows 系统下,一种常用的替代方法是多文档界面(multiple document interface,MDI),这种应用程序只有一个单一的主窗口,但可以对主窗口中央区域的多个文档窗口进行管理。Qt 可以在它支持的所有平台上创建 SDI 和 MDI 应用程序。图 3.17 给出了使用这两种方法的 Spreadsheet 应用程序。第 6 章将对 MDI 进行说明。

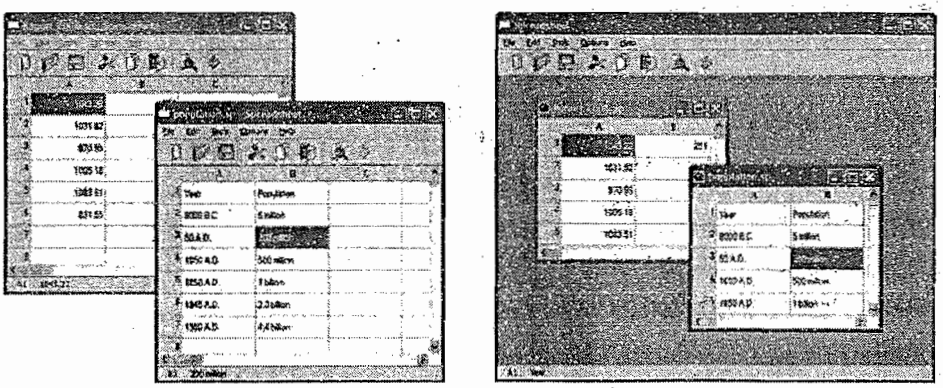


图 3.17 单文档界面和多文档界面

3.8 程序启动画面

许多应用程序都会在启动的时候显示一个程序启动画面(splash screen),图 3.18 给出的就是这

样的一个实例。一些程序员使用程序启动画面对缓慢的启动过程进行掩饰,而另外一些人则是用于满足市场部门的要求。使用 QSplashScreen 类,可以非常容易地为 Qt 应用程序添加一个程序启动画面。

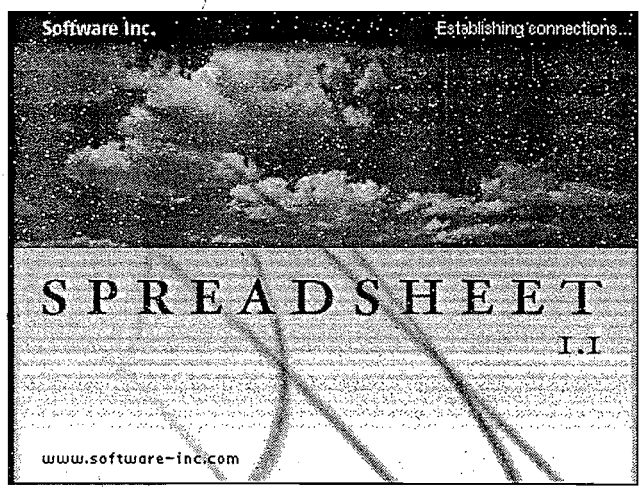


图 3.18 程序启动画面

类 QSplashScreen 会在应用程序的主窗口出现之前显示一个图片。它也可以在这个图片上显示一些消息,用来通知用户有关应用程序初始化的过程。通常,程序启动画面的代码会放在main()函数中,位于 QApplication::exec()调用之前。

下面给出了一个 main()函数的例子,在应用程序中,它使用 QSplashScreen 显示的程序启动画面表示启动时载入的一些模块和网络连接的建立。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QSplashScreen *splash = new QSplashScreen;
    splash->setPixmap(QPixmap(":/images/splash.png"));
    splash->show();

    Qt::Alignment topRight = Qt::AlignRight | Qt::AlignTop;

    splash->showMessage(QObject::tr("Setting up the main window..."),
                        topRight, Qt::white);
    MainWindow mainWin;

    splash->showMessage(QObject::tr("Loading modules..."),
                        topRight, Qt::white);
    loadModules();

    splash->showMessage(QObject::tr("Establishing connections..."),
                        topRight, Qt::white);
    establishConnections();

    mainWin.show();
    splash->finish(&mainWin);
    delete splash;

    return app.exec();
}
```

到此为止,我们已经创建了 Spreadsheet 应用程序的用户界面。在下一章中,将会通过实现电子制表软件的核心功能来完成这个应用程序。

第 4 章　实现应用程序的功能

前两章说明了如何创建 Spreadsheet 应用程序的用户界面。在这一章中，将通过编写它的底层功能函数来完成这个程序。此外，还将看到如何载入和保存文件，如何在内存中存储数据，如何实现剪贴板操作，以及如何向 QTableWidget 中添加对电子制表软件公式的支持等功能。

4.1　中央窗口部件

QMainWindow 的中央区域可以被任意种类的窗口部件所占用。下面给出的是对所有可能情形的概述。

1. 使用一个标准的 Qt 窗口部件

像 QTableWidget 或者 QTextEdit 这样的标准窗口部件可以用作中央窗口部件。在这种情况下，这个应用程序的功能，如文件的载入和保存，必须在其他地方实现（例如，在 QMainWindow 的子类中）。

2. 使用一个自定义窗口部件

特殊的应用程序通常需要在自定义窗口部件中显示数据。例如，一个图标编辑器程序就应当使用一个 IconEditor 窗口部件作为自己的中央窗口部件。第 5 章将会说明如何在 Qt 中编写自定义窗口部件。

3. 使用一个带布局管理器的普通 QWidget

有时，应用程序的中央区域会被许多窗口部件所占用。这时可以通过使用一个作为所有这些其他窗口部件父对象的 QWidget，以及通过使用布局管理器管理这些子窗口部件的大小和位置来完成这一特殊情况。

4. 使用切分窗口（splitter）

多个窗口部件一起使用的另一种方法是使用 QSplitter。QSplitter 会在水平方向或者竖直方向上排列它的子窗口部件，用户可以利用切分条（splitter handle）控制它们的尺寸大小。切分窗口可以包含所有类型的窗口部件，包括其他切分窗口。

5. 使用多文档界面工作空间

如果应用程序使用的是多文档界面，那么它的中央区域就会被 QMdiArea 窗口部件所占据，并且每个多文档界面窗口都是它的一个子窗口部件。

布局、切分窗口和多文档界面工作空间都可以与标准的 Qt 窗口部件或者自定义窗口部件组合使用。第 6 章将会进一步深入地讲解这些类。

对于 Spreadsheet 应用程序，会使用一个 QTableWidget 子类作为它的中央窗口部件。类 QTableWidget 已经提供了我们所需要的绝大多数电子制表软件的功能，但是它还不支持剪贴板操作，并且也不能理解诸如“ = A1+A2+A3”这样的电子制表软件公式的意义。我们将会在 Spreadsheet 类中实现这些缺少的功能。

4.2 子类化 QTableWidget

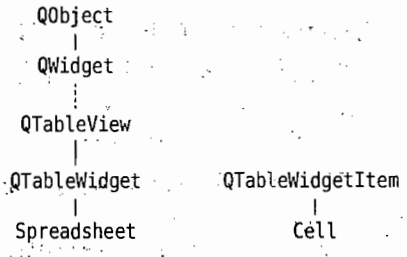


图 4.1 类 Spreadsheet 和 Cell 的继承树

类 Spreadsheet 派生自 QTableWidget,如图 4.1 所示。QTableWidget 是一组格子,可以非常有效地用来表达二维稀疏数组。它可以在规定的维数内显示用户滚动到的任一单元格。当用户在一个空单元格内输入一些文本的时候,QTableWidget 会自动创建一个用来存储这些文本的 QTableWidgetItem。

QTableWidget 派生自 QTableView,它是模型/视图类之一,我们将在第 10 章进一步了解它。对于另外一个表 QicsTable,它有更多的非常规功能,可以从 http://www.ics.com/中获取。

让我们一起来实现 Spreadsheet,首先从它的头文件开始:

```
#ifndef SPREADSHEET_H
#define SPREADSHEET_H

#include <QTableWidget>

class Cell;
class SpreadsheetCompare;
```

头文件是从 Cell 和 SpreadsheetCompare 类的前置声明开始的。

QTableWidget 单元格的属性,比如它的文本和对齐方式等,都存储在 QTableWidgetItem 中。与 QTableWidget 不同的是,QTableWidgetItem 不是一个窗口部件类,而是一个纯粹的数据类。Cell 类派生自 QTableWidgetItem,会在本章的最后一节对这个 Cell 类进行解释。

```
class Spreadsheet : public QTableWidget
{
    Q_OBJECT

public:
    Spreadsheet(QWidget *parent = 0);

    bool autoRecalculate() const { return autoRecalc; }
    QString currentLocation() const;
    QString currentFormula() const;
    QTableWidgetSelectionRange selectedRange() const;
    void clear();
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    void sort(const SpreadsheetCompare &compare);
```

之所以把 autoRecalculate()函数实现为内联函数,是因为无论自动重新计算的标识符生效与否,它都必须要有返回值。

在第 3 章中,当实现 MainWindow 时,我们依赖于 Spreadsheet 中的一些公有函数。例如,我们从 MainWindow::newFile()中调用 clear()来重置电子制表软件。也使用了一些从 QTableWidget 中继承而来的函数,特别是 setCurrentCell()和 setShowGrid()。

```
public slots:
    void cut();
    void copy();
    void paste();
    void del();
    void selectCurrentRow();
    void selectCurrentColumn();
    void recalculate();
    void setAutoRecalculate(bool recalc);
    void findNext(const QString &str, Qt::CaseSensitivity cs);
```

```
signals:
    void modified();
```

Spreadsheet 提供了许多实现 Edit、Tools 和 Options 菜单中的动作的槽，并且它也提供了一个 modified()信号，用来告知用户可能已经发生的任何变化。

```
private slots:
    void somethingChanged();
```

还定义了一个由 Spreadsheet 类内部使用的私有槽：

```
private:
    enum { MagicNumber = 0x7F51C883, RowCount = 999, ColumnCount = 26 };

    Cell *cell(int row, int column) const;
    QString text(int row, int column) const;
    QString formula(int row, int column) const;
    void setFormula(int row, int column, const QString &formula);

    bool autoRecalc;
};
```

在这个类的私有段中，声明了 3 个常量、4 个函数和 1 个变量。

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1,
                    const QStringList &row2) const;

    enum { KeyCount = 3 };
    int keys[KeyCount];
    bool ascending[KeyCount];
};

#endif
```

在这个头文件的最后，给出了 SpreadsheetCompare 类的定义。当查看 Spreadsheet::sort()时，会解释这个类。

现在来看一下它的实现文件：

```
#include <QtGui>

#include "cell.h"
#include "spreadsheet.h"

Spreadsheet::Spreadsheet(QWidget *parent)
    : QTableWidget(parent)
{
    autoRecalc = true;

    setItemPrototype(new Cell);
    setSelectionMode(ContiguousSelection);

    connect(this, SIGNAL(itemChanged(QTableWidgetItem *)),
            this, SLOT(somethingChanged()));

    clear();
}
```

通常情况下，当用户在一个空单元格中输入一些文本的时候，QTableWidget 将会自动创建一个 QTableWidgetItem 来保存这些文本。在电子制表软件中，我们想利用将要创建的 Cell 项来代替 QTableWidgetItem。这可以通过在构造函数中调用 setItemPrototype()来完成。实际上，QTableWidget 会在每次需要新项的时候把所传递的项以原型的形式克隆出来。

同样是在构造函数中，我们将选择模式设置为 QAbstractItemView::ContiguousSelection，从而可以允许简单矩形选择框方法。我们把表格窗口部件的 itemChanged()信号连接到私有槽 something-

Changed()上,这可以确保在用户编辑一个单元格的时候,somethingChanged()槽可以得到调用。最后,调用 clear()来重新调整表格的尺寸大小并且设置列标题。

```
void Spreadsheet::clear()
{
    setRowCount(0);
    setColumnCount(0);
    setRowCount(RowCount);
    setColumnCount(ColumnCount);

    for (int i = 0; i < ColumnCount; ++i) {
        QTableWidgetItem *item = new QTableWidgetItem;
        item->setText(QString(QChar('A' + i)));
        setHorizontalHeaderItem(i, item);
    }

    setCurrentCell(0, 0);
}
```

clear()函数是从 Spreadsheet 构造函数中得到调用的,用来初始化电子制表软件。它也会在 MainWindow::newFile()中得到调用。

我们原本使用 QTableWidget::clear()来清空所有项和任意选择,但是那样做的话,这些标题将会以当前大小的尺寸而被留下。相反的是,我们要把表格向下调整为 0×0。这样就可以完全清空整个表格,包括这些标题。然后,重新调整表的大小为 ColumnCount× RowCount(26×999),并且把 QTableWidgetItem 水平方向上的标题修改为列名"A","B",…,"Z"。不需要设置垂直标题的标签,因为这些标签的默认值是"1","2",…,"999"。最后,把单元格光标移动到单元格 A1 处。

QTableWidget 由多个子窗口部件构成。在它的顶部有一个水平的 QHeaderView,左侧有一个垂直的 QHeaderView,还有两个 QScrollBar。在它的中间区域被一个名为视口(viewport)的特殊窗口部件所占用,QTableWidget 可以在它上面绘制单元格。通过从 QTableView 和 QAbstractScrollArea 中继承的一些函数,可以访问这些不同的子窗口部件(参见图 4.2)。QAbstractScrollArea 提供了一个可以滚动的视口和两个可以打开或关闭的滚动条。第 6 章将讲述 QScrollArea 子类。

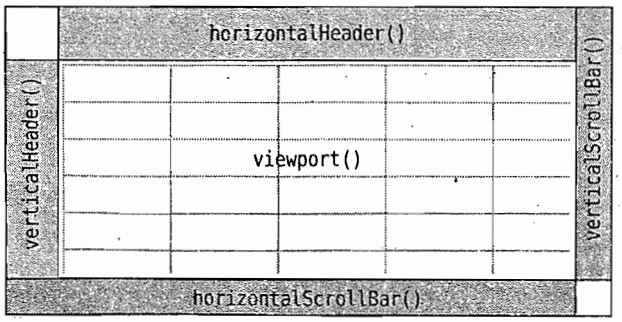


图 4.2 构成 QTableWidget 的各个窗口部件

```
Cell *Spreadsheet::cell(int row, int column) const
{
    return static_cast<Cell *>(item(row, column));
}
```

cell()私有函数可以根据给定的行和列返回一个 Cell 对象。它几乎和 QTableWidget::item()函数的作用一样,只不过它返回的是一个 Cell 指针,而不是一个 QTableWidgetItem 指针。

```
QString Spreadsheet::text(int row, int column) const
{
    Cell *c = cell(row, column);
```

```
    if (c) {
        return c->text();
    } else {
        return "";
    }
}
```

text()私有函数可以返回给定单元格中的文本。如果 cell()返回的是一个空指针,则表示该单元格是空的,因而返回一个空字符串。

```
QString Spreadsheet::formula(int row, int column) const
{
    Cell *c = cell(row, column);
    if (c) {
        return c->formula();
    } else {
        return "";
    }
}
```

formula()函数返回给定单元格中的公式。在很多情况下,公式和文本是相同的。例如,公式"Hello"等价于字符串"Hello",所以如果用户在单元格中输入"Hello"并且按下回车键,那么该单元格就会显示文本"Hello"。但是还有一些例外的情况:

- 如果公式是一个数字,那么它就会被认为是一个数字。例如,公式"1.50"等价于双精度实数(double)的 1.5,它在电子制表软件中会被显示为右对齐的"1.5"。
- 如果公式以单引号开始,那么公式的剩余部分将会被认为是文本。例如,公式"'12345"等价于字符串"12345"。
- 如果公式以等号开始,那么公式将会被认为是一个算术公式。例如,如果单元格 A1 包含"12"并且单元格 A2 包含"6",那么公式"=A1+A2"就会等于 18。

把公式转换成值的任务是由 Cell 类完成的。这时,要记住的事情是显示在单元格内的文本是公式的结果,而不是公式本身。

```
void Spreadsheet::setFormula(int row, int column,
                             const QString &formula)
{
    Cell *c = cell(row, column);
    if (!c) {
        c = new Cell;
        setItem(row, column, c);
    }
    c->setFormula(formula);
}
```

setFormula()私有函数可以设置用于给定单元格的公式。如果该单元格已经有一个 Cell 对象,那么我们就重新使用它。否则,可以创建一个新的 Cell 对象并且调用 QTableWidget::setItem()把它插入到表中。最后,调用该单元格自己的 setFormula()函数,但如果这个单元格已经显示在屏幕上,那么就重新绘制它。我们不需要担心随后对这个 Cell 对象的删除操作,因为 QTableWidget 会得到这个单元格的所有权,并且会在正确的时候自动将其删除。

```
QString Spreadsheet::currentLocation() const
{
    return QChar('A' + currentColumn())
           + QString::number(currentRow() + 1);
}
```

currentLocation()函数返回当前单元格的位置,它是按照电子制表软件的通常格式,也就是一个

列字母后跟上行号的形式来表示这个位置的值。MainWindow::updateStatusBar()使用它把这个单元格的位置显示在状态栏上。

```
QString Spreadsheet::currentFormula() const
{
    return formula(currentRow(), currentColumn());
}
```

currentFormula()函数返回当前单元格的公式。它是从 MainWindow::updateStatusBar()中得到调用的。

```
void Spreadsheet::somethingChanged()
{
    if (autoRecalc)
        recalculate();
    emit modified();
}
```

如果启用了"auto-recalculate"(自动重新计算),那么 somethingChanged()私有槽就会重新计算整个电子制表软件。它也会发射 modified()信号。

把数据存储为项

在 Spreadsheet 应用程序中,每一个非空单元格都被当作一个独立的 QTableWidgetItem 对象而保存到内存中。把数据存储为项(item)是一种对 QListWidgetItem 和 QTreeWidgetItem 进行操作的方法,该方法也可用于 QListWidget 和 QTreeWidget。

Qt 的项类可以用作非常规的数据持有者。例如,一个 QTableWidgetItem 已经储存了一些属性,其中包括一个字符串、一种字体、一种颜色和一个图标,以及一个返回到 QTableWidget 的指针。项也可以保存数据(QVariant 型),包括一些已经注册过的自定义类型,以及通过项类的子类化,我们还可以提供其他功能。

许多老一点的工具包在它们的项类中提供一个 void 指针来储存自定义数据。在 Qt 中,更为自然的方法是使用带 QVariant 的 setData(),但如果需要一个 void 指针,那么可以通过子类化一个项类并且添加一个 void 指针成员变量来很简单地实现这一点。

对于更具有挑战性的数据处理需求,比如大数据集、复杂数据项、数据库集成以及多数据视图等,Qt 提供了一套模型/视图(model/view)类,利用这些类可以把数据从它们的直观表示中分离出来。这些内容将会在第 10 章中加以讲述。

4.3 载入和保存

现在,我们将使用一种自定义的二进制数格式来实现 Spreadsheet 文件的载入和保存。将使用 QFile 和 QDataStream 来完成这一工作,由它们共同提供与平台无关的二进制数输入/输出接口。

首先从一个 Spreadsheet 文件的输出开始:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::WriteOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                             tr("Cannot write file %1:\n%2.")
                             .arg(file.fileName())
                             .arg(file.errorString()));
```

```
        return false;
    }

    QDataStream out(&file);
    out.setVersion(QDataStream::Qt_4_3);

    out << quint32(MagicNumber);

    QApplication::setOverrideCursor(Qt::WaitCursor);
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

从 MainWindow::saveFile()中调用的 writeFile()函数把文件输出到磁盘中。如果输出成功，它会返回 true；如果出现错误，则返回 false。

我们使用给定的文件名创建一个 QFile 对象，并且调用 open()打开这个用于输出的文件。我们也会创建一个 QDataStream 对象，由它操作这个 QFile 对象并且使用该对象输出数据。

在输出数据之前，我们把这个应用程序的光标修改为标准的等待光标(通常是一个沙漏)，并且一旦所有的数据输出完毕，就需要把这个应用程序的光标重新恢复为普通光标。在函数的最后，文件会由 QFile 对象的析构函数自动关闭。

QDataStream 既可以支持 C++ 基本类型，也可以支持多种 Qt 类型。该语法模仿了标准 C++ 的 <iostream> 中的那些类的语法。例如：

```
out << x << y << z;
```

会把变量 x、y 和 z 输出到一个流中，而：

```
in >> x >> y >> z;
```

会从流中读出它们。因为 C++ 的基本类型在不同平台上可能会有不同的大小，所以把这些变量强制转换成 qint8、quint8、qint16、quint16、qint32、quint32、qint64 以及 quint64 中的一个是最安全的做法，这样做可以确保它们能够获得应有的大小(按位计算)。

Spreadsheet 应用程序的文件格式是相当简单的。一个 Spreadsheet 文件以一个 32 位数字作为文件的开始，由它确定文件的格式(MagicNumber，在 spreadsheet.h 中定义为 0x7F51C883，它是一个任意的随机数)。然后是连续的数据块，每一数据块都包含了用于一个单元格中的行、列和公式。为了节省空间，我们没有输出空白单元格。该文件格式如图 4.3 所示。



图 4.3　Spreadsheet 的文件格式

关于这些数据类型的二进制数确切表示方法则是由 QDataStream 决定的。例如，一个 quint16 按照高字节在后的顺序存储为两个字节，而一个 QString 则被存储为字符串的长度后跟 Unicode 字符的形式。

关于 Qt 数据类型的二进制数确切表示方法，自 Qt 1.0 以来已经发生了许多变化。而且在未来的 Qt 发行版中，为了能够与现存的数据类型和将来允许出现的新的 Qt 类型保持一致，这样的表示方法还可能会继续变化下去。默认情况下，QDataStream 会使用最近版本的二进制数格式(在 Qt 4.3 中

的版本是第 9 版),但是可以设置它,使它可以读取那些旧的数据版本。如果以后有可能使用新的 Qt 发行版来重新编译这个应用程序,那么为了避免出现任何可能的兼容性问题,需要明确告诉 QDataStream 应该使用的是第 9 版,从而无需再考虑要使用的 Qt 版本。(QDataStream::Qt_4_3 是一个方便的常量,它就等于 9。)

QDataStream 的功能非常齐全。既可以把它用于 QFile 中,也可以把它用于 QBuffer、QProcess、QTcpSocket、QUdpSocket 或者 QSslSocket 中。在读取和输出文本文件时,Qt 也提供了一个 QTextStream 类,可以使用它代替 QDataStream 类。第 12 章将深入地讲解这些类,并且也会讲述处理不同的 QDataStream 版本时所使用的各种方法。

```
bool Spreadsheet::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                             tr("Cannot read file %1:\n%2.")
                             .arg(file.fileName())
                             .arg(file.errorString()));
        return false;
    }

    QDataStream in(&file);
    in.setVersion(QDataStream::Qt_4_3);

    quint32 magic;
    in >> magic;
    if (magic != MagicNumber) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                             tr("The file is not a Spreadsheet file."));
        return false;
    }

    clear();

    quint16 row;
    quint16 column;
    QString str;

    QApplication::setOverrideCursor(Qt::WaitCursor);
    while (!in.atEnd()) {
        in >> row >> column >> str;
        setFormula(row, column, str);
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

readFile()函数与 writeFile()函数非常相似。我们使用 QFile 读取一个文件,但这一次使用的是 QIODevice::ReadOnly 标记,而不是 QIODevice::WriteOnly 标记。然后,把 QDataStream 的版本设置为 9。用于读取文件的格式必须总是与输出文件的格式相同。

如果该文件在开始处具有正确的幻数(magic number),那么可以调用 clear()来清空电子制表软件中的所有单元格,并且读入单元格中的数据。由于该文件中只包含那些非空单元格的数据,并且也不大可能重置电子制表软件中的每个单元格,所以必须确保在读入数据之前已经清空了所有的单元格。

4.4 实现 Edit 菜单

现在,我们已经为实现响应应用程序 Edit 菜单中的各个槽做好了准备。Spreadsheet 应用程序中的 Edit 菜单如图 4.4 所示。

```
void Spreadsheet::cut()
{
    copy();
    del();
}
```

cut()槽可以对 Edit→Cut 菜单做出响应。由于 Cut 的执行效果与 Copy 之后再加上一个 Delete 的执行效果相同,所以其实现代码很简单。

```
void Spreadsheet::copy()
{
    QTableWidgetSelectionRange range = selectedRange();
    QString str;

    for (int i = 0; i < range.rowCount(); ++i) {
        if (i > 0)
            str += "\n";
        for (int j = 0; j < range.columnCount(); ++j) {
            if (j > 0)
                str += "\t";
            str += formula(range.topRow() + i, range.leftColumn() + j);
        }
    }
    QApplication::clipboard()->setText(str);
}
```

copy()槽能够对 Edit→Copy 做出响应。它会遍历当前选择(如果没有明确的选择,那么就认为选择的只是当前单元格)。每一个选中单元格的公式都会被添加到一个 QString 中,行与行之间利用换行符"\n"分隔,列与列之间则以制表符"\t"来分隔。图 4.5 给出了这一实现方法的示意图。

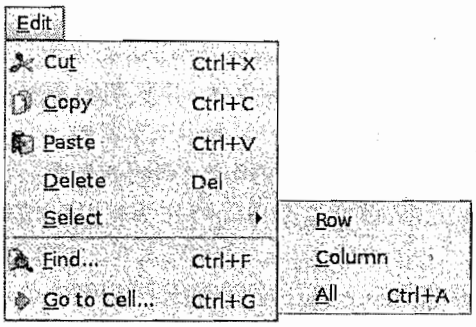


图 4.4　Spreadsheet 应用程序的 Edit 菜单

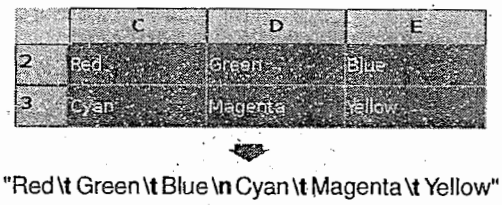


图 4.5　把选择复制到剪贴板中

在 Qt 中,通过调用 QApplication::clipboard()静态函数可以使用系统的剪贴板。通过调用 QClipboard::setText(),就既可以在本应用程序中又可以在其他应用程序中使用放在剪贴板上的这些文本。这种使用制表符"\t"和换行符"\n"作为文件分隔符的形式可以被包括微软 Excel 在内的许多应用程序所支持。

函数 QTableWidget::selectedRanges()返回一个选择范围列表。我们知道,由于在构造函数中已经将选择模式设置为 QAbstractItemView::ContiguousSelection,所以选择范围不可能再超过 1。为方便起见,我们定义了一个 selectedRange()函数来返回这个选择范围:

```
QTableWidgetSelectionRange Spreadsheet::selectedRange() const
{
    QList<QTableWidgetSelectionRange> ranges = selectedRanges();
    if (ranges.isEmpty())
        return QTableWidgetSelectionRange();
    return ranges.first();
}
```

如果只有一个选择,则只需简单地返回第一个(并且也只有这一个)选择即可。没有选择的情

况应该永远不会发生,因为 ContiguousSelection 模式至少可以把当前单元格当作是已经选中的选择。但是,为了避免使程序出现缺陷的可能性,还是需要对这种当前没有选中单元格的情况进行单独处理。

```
void Spreadsheet::paste()
{
    QTableWidgetSelectionRange range = selectedRange();
    QString str = QApplication::clipboard()->text();
    QStringList rows = str.split('\n');
    int numRows = rows.count();
    int numColumns = rows.first().count('\t') + 1;

    if (range.rowCount() * range.columnCount() != 1
            && (range.rowCount() != numRows
                || range.columnCount() != numColumns)) {
        QMessageBox::information(this, tr("Spreadsheet"),
                tr("The information cannot be pasted because the copy "
                   "and paste areas aren't the same size."));
        return;
    }

    for (int i = 0; i < numRows; ++i) {
        QStringList columns = rows[i].split('\t');
        for (int j = 0; j < numColumns; ++j) {
            int row = range.topRow() + i;
            int column = range.leftColumn() + j;
            if (row < RowCount && column < ColumnCount)
                setFormula(row, column, columns[j]);
        }
    }
    somethingChanged();
}
```

paste()槽对 Edit→Paste 菜单选项做出响应。我们从剪贴板中取回文本,并且调用静态函数 QString::split()把这串字符变成一个 QStringList。每行都会变成这个列表中的一个字符串。

接下来,需要求出复制区域的维数。行数就是 QStringList 中字符串的个数,列数就是第一行中制表符"\t"字符的个数再加上 1。如果只选中了一个单元格,就把这个单元格作为粘贴区域放在左上角;否则,就把当前选择作为要粘贴的区域。

为了执行粘贴操作,我们遍历所有行并且再次使用 QString::split()把它们分隔到每一个单元格中,但是这一次要把制表符"\t"当作分隔符。图 4.6 给出了这一过程中所使用的步骤。

```
void Spreadsheet::del()
{
    QList<QTableWidgetItem *> items = selectedItems();
    if (!items.isEmpty()) {
        foreach (QTableWidgetItem *item, items)
            delete item;
        somethingChanged();
    }
}
```

del()槽对 Edit→Delete 菜单选项做出响应。如果有选中的项,那么该函数就会删除它们并且调用 somethingChanged()函数。对选择中的每一个 Cell 对象使用 delete 足以清空所有这些单元格。当删除 QTableWidget 的 QTableWidgetItem 的时候,QTableWidget 就会注意到这一情况的发生,而如果这些项中有可见的任意项,QTableWidget 将会自动对自己进行重绘。如果在一个已经删除过的单元格位置上又调用了 cell(),那么该函数将会返回一个空指针。

```
void Spreadsheet::selectCurrentRow()
{
    selectRow(currentRow());
}
```

```
void Spreadsheet::selectCurrentColumn()
{
    selectColumn(currentColumn());
}
```

selectCurrentRow()和 selectCurrentColumn()对 Edit→Select→Row 和 Edit→Select→Column 菜单选项做出响应。这些实现分别依赖于 QTableWidget 的 selectRow()和 selectColumn()函数。我们不必再去实现 Edit→Select→All 菜单选项的功能，因为该功能可以由 QTableWidget 从 QAbstractItemView::selectAll()的函数中继承过来。

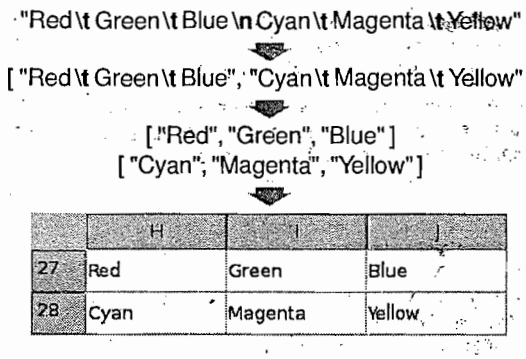


图 4.6　把剪贴板中的文本粘贴到电子制表软件中

```
void Spreadsheet::findNext(const QString &str, Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() + 1;

    while (row < RowCount) {
        while (column < ColumnCount) {
            if (text(row, column).contains(str, cs)) {
                clearSelection();
                setCurrentCell(row, column);
                activateWindow();
                return;
            }
            ++column;
        }
        column = 0;
        ++row;
    }
    QApplication::beep();
}
```

findNext()槽会遍历单元格一遍，它从当前光标右侧的单元格开始遍历到这一行的最后一列，然后再从下一行的第一个单元格开始继续遍历，如此反复，直到找到所要查找的文本，或者是直到最后一个单元格为止。例如，如果当前的单元格是 C24，那么就会搜索 D24、E24、…、Z24，然后再去搜索 A25、B25、C25、…、Z25，等等，一直遍历到 Z999 为止。如果找到了一个匹配项，那么就清空当前选择，把单元格光标移动到那个匹配的单元格上，并且让包含 Spreadsheet 的窗口变成激活状态。如果没能找到匹配的单元格，那么就让应用程序发出“哔”(beep)的一声来表明搜索已经结束，匹配没有成功。

```
void Spreadsheet::findPrevious(const QString &str,
                               Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() - 1;

    while (row >= 0) {
```

```
        while (column >= 0) {
            if (text(row, column).contains(str, cs)) {
                clearSelection();
                setCurrentCell(row, column);
                activateWindow();
                return;
            }
            --column;
        }
        column = ColumnCount - 1;
        --row;
    }
    QApplication::beep();
}
```

findPrevious()槽与 findNext()槽相似,区别之处是它会向相反的方向遍历并且会在单元格 A1 处停下来。

4.5 实现其他菜单

现在,我们将要实现那些对 Tools 和 Options 菜单做出响应的槽。这些菜单项如图 4.7 所示。

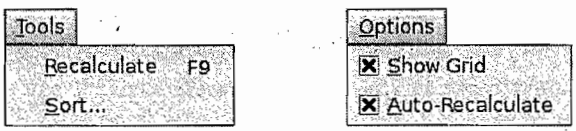


图 4.7 Spreadsheet 应用程序的 Tools 和 Options 菜单

```
void Spreadsheet::recalculate()
{
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            if (cell(row, column))
                cell(row, column)->setDirty();
        }
    }
    viewport()->update();
}
```

recalculate()槽能够对 Tools→Recalculate 菜单选项做出响应。当必要时,它也会被 Spreadsheet 自动调用。

我们遍历每一个单元格,并且对每一个单元格调用 setDirty()把它们标记为需要重新计算。为了在电子制表软件中显示一个 Cell 对象的值,QTableWidget 会再次对该对象调用 text()以获得其值,从而使该值重新计算一次。

然后,对这个视口调用 update()来重新绘制整个电子制表软件。QTableWidget 中的重绘代码就又会对每一个可见单元格调用 text()来获得它们中要显示的值。因为在每一个单元格上都调用了 setDirty(),所以这些对 text()的调用将会使用重新计算过的值。该计算可能需要重新计算那些不可见的单元格,这就会造成一个级联计算,直到每一个需要被重新计算的单元格能够在刚才刷新过的视口中重新得到计算,从而使它们也能够显示正确的文本。这一计算是由 Cell 类执行的。

```
void Spreadsheet::setAutoRecalculate(bool recalc)
{
    autoRecalc = recalc;
    if (autoRecalc)
        recalculate();
}
```

setAutoRecalculate()槽对 Options→Auto-Recalculate 菜单选项做出响应。如果启用了这个特性，则会立即重新计算整个电子制表软件以确保它是最新的，然后，recalculate()会自动在 somethingChanged()中得到调用。

因为 QTableWidget 已经提供了一个从 QTableView 中继承而来的 setShowGrid()槽，所以不需要再对 Options→Show Grid 菜单选项编写任何代码。所有要保留的东西就是 Spreadsheet::sort()，它会在 MainWindow::sort()中得到调用：

```
void Spreadsheet::sort(const SpreadsheetCompare &compare)
{
    QList<QStringList> rows;
    QTableWidgetSelectionRange range = selectedRange();
    int i;

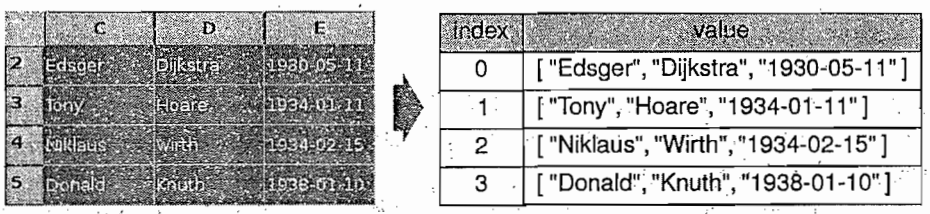
    for (i = 0; i < range.rowCount(); ++i) {
        QStringList row;
        for (int j = 0; j < range.columnCount(); ++j)
            row.append(formula(range.topRow() + i,
                               range.leftColumn() + j));
        rows.append(row);
    }

    qStableSort(rows.begin(), rows.end(), compare);

    for (i = 0; i < range.rowCount(); ++i) {
        for (int j = 0; j < range.columnCount(); ++j)
            setFormula(range.topRow() + i, range.leftColumn() + j,
                       rows[i][j]);
    }

    clearSelection();
    somethingChanged();
}
```

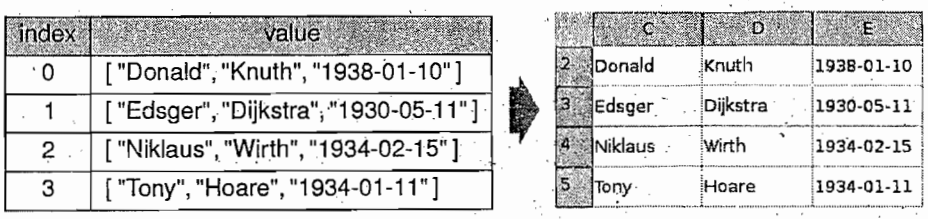
排序操作会对当前的选择进行，并且会根据存储在 compare 对象中的排序键和排序顺序重新排列这些行。我们使用一个 QStringList 来重新表示每一行数据，并且把该选择存储在一个行列表中。我们使用 Qt 的 qStableSort()算法，并且根据公式而不是根据值来进行简单排序。这一过程如图 4.8 和图 4.9 所示。第 11 章中会讲述 Qt 的标准算法和数据结构。



	C	D	E
2	Edsger	Dijkstra	1930-05-11
3	Tony	Hoare	1934-01-11
4	Niklaus	Wirth	1934-02-15
5	Donald	Knuth	1938-01-10

index	value
0	["Edsger", "Dijkstra", "1930-05-11"]
1	["Tony", "Hoare", "1934-01-11"]
2	["Niklaus", "Wirth", "1934-02-15"]
3	["Donald", "Knuth", "1938-01-10"]

图 4.8　把选择存储为一个行列表



index	value
0	["Donald", "Knuth", "1938-01-10"]
1	["Edsger", "Dijkstra", "1930-05-11"]
2	["Niklaus", "Wirth", "1934-02-15"]
3	["Tony", "Hoare", "1934-01-11"]

	C	D	E
2	Donald	Knuth	1938-01-10
3	Edsger	Dijkstra	1930-05-11
4	Niklaus	Wirth	1934-02-15
5	Tony	Hoare	1934-01-11

图 4.9　排序后把数据放回表中

qStableSort()函数可以接受一个开始迭代器、一个终止迭代器和一个比较函数。这个比较函数是一个带两个参数(两个 QStringList)的函数，并且如果第一个参数“小于”第二个参数，它就返回

true,否则返回 false。传递的作为比较函数的这个 compare 对象并不是一个真正的函数,但是它可以用作一个函数,将会很快看到这一点。

在执行完 qStableSort()之后,我们把数据移回到这个表中,接着清空这一选择,并且调用 somethingChanged()函数。

在 spreadsheet.h 文件中,SpreadsheetCompare 类的定义如下:

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1,
                    const QStringList &row2) const;

    enum { KeyCount = 3 };
    int keys[KeyCount];
    bool ascending[KeyCount];
};
```

SpreadsheetCompare 类有些特殊,因为它实现了一个"()"操作符。这样就允许把这个类像函数一样使用。把这样的类称为函数对象(function object),或者称为仿函数(functor)。为了理解仿函数是如何工作的,首先从一个简单的例子开始:

```
class Square
{
public:
    int operator()(int x) const { return x * x; }
}
```

Square 类提供了一个函数,operator()(int)函数,它返回其参数的平方值。通过把这个函数命名为 operator()(int),而不是将其命名为 compute(int)之类的函数,就可以把一个类型为 Square 的对象当作一个函数。

```
Square square;
int y = square(5);
// y equals 25
```

现在,让我们来看一个包括 SpreadsheetCompare 的实例:

```
QStringList row1, row2;
SpreadsheetCompare compare;
...
if (compare(row1, row2)) {
    // row1 is less than row2
}
```

使用 compare 对象就像使用一个普通的 compare()函数一样。另外,它的实现可以访问所有存储为成员变量的排序键和排序顺序。

与此方案相同的另一种方法是,把这些排序键和排序顺序存储在全局成员变量中,并且使用一个普通的 compare()函数。然而,在全局成员变量之间通信是一种并不提倡的做法,并且可能会产生一些莫名其妙的问题。作为像 qStableSort()这样的模板函数的接口,仿函数是一种更为常用的做法。

这里给出的是对电子制表软件中的两个行进行比较的函数实现:

```
bool SpreadsheetCompare::operator()(const QStringList &row1,
                                    const QStringList &row2) const
{
    for (int i = 0; i < KeyCount; ++i) {
        int column = keys[i];
        if (column != -1) {
            if (row1[column] != row2[column]) {
                if (ascending[i]) {
                    return row1[column] < row2[column];
```

```
                } else {
                    return row1[column] > row2[column];
                }
            }
        }
    }
    return false;
}
```

如果第 1 行小于第 2 行，该仿函数就返回 true；否则，就返回 false。qStableSort()函数会使用这个函数的结果来执行排序操作。

SpreadsheetCompare 对象的 key 与 ascending 数组和 MainWindow::sort()函数（已经在第 2 章中给出过）一起配合使用。每个键都保存一个列索引，或者 -1（为“None”时）。

我们按键顺序比较两行中相应的单元格条目。一旦发现有不同之处，就返回一个适当的 true 或者 false 值。如果所有的比较关系都证明两者是相等的，就返回 false。qStableSort()函数会使用这里给出的顺序来解决这种平局情形。如果一开始的时候 row1 在 row2 之前，并且它们都不“小于”对方，那么，在结果中 row1 还在 row2 前面。这就是 qStableSort()与它很相似的非稳定版本的 qSort()函数之间的区别。

现在已经完成了这个 Spreadsheet 类。在下一节中，将分析 Cell 类的代码。这个类用作保存单元格的公式，并且它还重新实现了 QTableWidgetItem::data()函数，Spreadsheet 可以通过 QTableWidgetItem::text()间接调用该函数，用它显示单元格公式的计算结果。

4.6　子类化 QTableWidgetItem

Cell 类派生自 QTableWidgetItem 类。这个类被设计用于和 Spreadsheet 一起工作，但是它对类 QTableWidgetItem 没有任何特殊的依赖关系，所以在理论上讲，它也可以用于任意的 QTableWidget 类中。这里给出的是 Cell 类的头文件：

```
#ifndef CELL_H
#define CELL_H

#include <QTableWidgetItem>

class Cell : public QTableWidgetItem
{
public:
    Cell();

    QTableWidgetItem *clone() const;
    void setData(int role, const QVariant &value);
    QVariant data(int role) const;
    void setFormula(const QString &formula);
    QString formula() const;
    void setDirty();

private:
    QVariant value() const;
    QVariant evalExpression(const QString &str, int &pos) const;
    QVariant evalTerm(const QString &str, int &pos) const;
    QVariant evalFactor(const QString &str, int &pos) const;

    mutable QVariant cachedValue;
    mutable bool cacheIsDirty;
};

#endif
```

通过增加两个私有变量，Cell 类对 QTableWidgetItem 进行了扩展：

- cachedValue 把单元格的值缓存为 QVariant。
- 如果缓存的值不是最新的,那么就把 cacheIsDirty 设置为 true。

之所以使用 QVariant,是因为有些单元格是 double 型值,另外一些单元格则是 QString 型值。

在声明 cachedValue 和 cacheIsDirty 变量时使用了 C++ 的 mutable 关键字,这样就可以在 const 函数中修改这些变量。或者,在每次调用 text()时,本应当重新计算这个值,但是这样做是不必要的,因为它的效率非常低下。

我们注意到,在该类的定义中并没有使用 Q_OBJECT 宏。这是因为,Cell 是一个普通的 C++ 类,它没有使用任何信号或者槽。实际上,因为 QTableWidgetItem 不是从 QObject 派生而来的,所以就不能让 Cell 拥有信号和槽。为了使 Qt 的项(item)类的开销降到最低,它们就不是从 QObject 派生的。如果需要信号和槽,可以在包含项的窗口部件中实现它们,或者在特殊情况下,可以通过对 QObject 进行多重继承的方式来实现它们。

以下是 cell.cpp 文件的开始部分:

```
#include <QtGui>

#include "cell.h"

Cell::Cell()
{
    setDirty();
}
```

在构造函数中,只需要将缓存设置为 dirty。没有必要传递父对象,当用 setItem()把单元格插入到一个 QTableWidget 中的时候,QTableWidget 将会自动对其拥有所有权。

每个 QTableWidgetItem 都可以保存一些数据,最多可以为每个数据"角色"分配一个 QVariant 变量。最常用的角色是 Qt::EditRole 和 Qt::DisplayRole。编辑角色用在那些需要编辑的数据上,而显示角色用在那些需要显示的数据上。通常情况下,用于两者的数据是一样的,但在 Cell 类中,编辑角色对应于单元格的公式,而显示角色对应于单元格的值(对公式求值后的结果)。

```
QTableWidgetItem *Cell::clone() const
{
    return new Cell(*this);
}
```

当 QTableWidget 需要创建一个新的单元格时,例如,当用户在一个以前没有使用过的空白单元格中开始输入数据时,它就会调用 clone()函数。传递给 QTableWidget::setItemPrototype()中的实例就是需要克隆的项。由于对于 Cell 来讲,成员级的复制已经足以满足需要,所以在clone()函数中,只需依靠由 C++ 自动创建的默认复制构造函数就可以创建新的 Cell 实例了。

```
void Cell::setFormula(const QString &formula)
{
    setData(Qt::EditRole, formula);
}
```

setFormula()函数用来设置单元格中的公式。它只是一个对编辑角色调用 setData()的简便函数。也可以从 Spreadsheet::setFormula()中调用它。

```
QString Cell::formula() const
{
    return data(Qt::EditRole).toString();
}
```

formula()函数会从 Spreadsheet::formula()中得到调用。就像 setFormula()一样,它也是一个简便函数,这次是重新获得该项的 EditRole 数据。

```
void Cell::setData(int role, const QVariant &value)
{
    QTableWidgetItem::setData(role, value);
    if (role == Qt::EditRole)
        setDirty();
}
```

如果有一个新的公式,就可以把 cacheIsDirty 设置为 true,以确保在下一次调用 text()的时候可以重新计算该单元格。

尽管对 Cell 实例中的 Spreadsheet::text()调用了 text(),但在 Cell 中没有定义 text()函数。这个 text()函数是一个由 QTableWidgetItem 提供的简便函数。这相当于调用 data(Qt::DisplayRole).toString()。

```
void Cell::setDirty()
{
    cacheIsDirty = true;
}
```

调用 setDirty()函数可以用来对该单元格的值强制进行重新计算。它只是简单地把 cacheIsDirty 设置为 true,也就意味着 cachedValue 不再是最新值了。除非有必要,否则不会执行这个重新计算操作。

```
QVariant Cell::data(int role) const
{
    if (role == Qt::DisplayRole) {
        if (value().isValid()) {
            return value().toString();
        } else {
            return "####";
        }
    } else if (role == Qt::TextAlignmentRole) {
        if (value().type() == QVariant::String) {
            return int(Qt::AlignLeft | Qt::AlignVCenter);
        } else {
            return int(Qt::AlignRight | Qt::AlignVCenter);
        }
    } else {
        return QTableWidgetItem::data(role);
    }
}
```

data()函数是从 QTableWidgetItem 中重新实现的。如果使用 Qt::DisplayRole 调用这个函数,那么它返回在电子制表软件中应该显示的文本;如果使用 Qt::EditRole 调用这个函数,那么它返回该单元格中的公式;如果使用 Qt::TextAlignmentRole 调用这个函数,那么它返回一个合适的对齐方式。在使用 DisplayRole 的情况下,它依靠 value()来计算单元格的值。如果该值是无效的(由于这个公式是错误的),则返回“####”。

在 data()中使用的这个 Cell::value()函数可以返回一个 QVariant 值。QVariant 可以存储不同类型的值,比如 double 和 QString,并且提供了把变量转换为其他类型变量的一些函数。例如,对一个保存了 double 值的变量调用 toString(),可以产生一个表示这个 double 值的字符串。使用默认构造函数构造的 QVariant 是一个“无效”变量。

```
const QVariant Invalid;

QVariant Cell::value() const
{
    if (cacheIsDirty) {
        cacheIsDirty = false;

        QString formulaStr = formula();
        if (formulaStr.startsWith('\'')) {
            cachedValue = formulaStr.mid(1);
```

```
        } else if (formulaStr.startsWith('=')) {
            cachedValue = Invalid;
            QString expr = formulaStr.mid(1);
            expr.replace(" ", "");
            expr.append(QChar::Null);

            int pos = 0;
            cachedValue = evalExpression(expr, pos);
            if (expr[pos] != QChar::Null)
                cachedValue = Invalid;
        } else {
            bool ok;
            double d = formulaStr.toDouble(&ok);
            if (ok) {
                cachedValue = d;
            } else {
                cachedValue = formulaStr;
            }
        }
    }
    return cachedValue;
}
```

value()私有函数返回这个单元格的值。如果 cacheIsDirty 是 true,就需要重新计算这个值。

如果公式是由单引号开始的(例如,“ '12345”),那么这个单引号就会占用位置 0,而值就是从位置 1 直到最后位置的一个字符串。

如果公式是由等号开始的,那么会使用从位置 1 开始的字符串,并且将它可能包含的任意空格全部移除。然后,调用 evalExpression()来计算这个表达式的值。这里的参数 pos 是通过引用(reference)方式传递的,由它来说明需要从哪里开始解析字符的位置。在调用 evalExpression()之后,如果表达式解析成功,那么在位置 pos 处的字符应当是我们添加上的 QChar::Null 字符。如果在表达式结束之前解析失败了,那么可以把 cachedValue 设置为 Invalid。

如果公式不是由单引号或者等号开始的,那么可以使用 toDouble()试着把它转换为浮点数。如果转换正常,就把 cachedValue 设置为结果数字;否则,把 cachedValue 设置为字符串公式。例如,公式“1.50”会导致 toDouble()把 ok 设置为 true 并且返回 1.5,而公式“World Population”则会导致 toDouble()把 ok 设置为 false 并且返回 0.0。

通过给 toDouble()一个 bool 指针,可以区分字符串转换中表示的是数字 0.0 还是表示的是转换错误(此时,仍旧会返回一个 0.0,但是同时会把这个 bool 设置为 false)。有时候,对于转换失败所返回的 0 值可能正是我们所需要的。在这种情况下,就没有必要再麻烦地传递一个 bool 指针了。考虑到程序的性能和移植性因素,Qt 从来不使用 C++ 异常(exception)机制来报告错误。但是,如果你的编译器支持 C++ 异常,那么这也不会妨碍你在自己的 Qt 程序中使用它们。

value()函数声明为 const 函数。我们不得不把 cachedValue 和 cacheIsValid 声明为 mutable 变量,以便编译器可以让我们在 const 函数中修改它们。当然,如果能够把 value()声明为一个非 const 函数并且移除 mutable 关键字可能会更吸引人些,但是这将会导致无法编译,因为是从一个 const 函数的 data()函数中调用 value()的。

除了要解析这些公式之外,现在已经完成了整个 Spreadsheet 应用程序。这一节的剩余部分将说明 evalExpression()以及 evalTerm()和 evalFactor()这两个帮助函数。这些代码有一些复杂,但是把它们放在这里是为了能够让整个应用程序显得更完善些。由于这些代码和图形用户界面编程无关,所以你可以非常放心地略过这一部分,从第 5 章继续阅读下去。

evalExpression()函数返回一个电子制表软件表达式的值。表达式可以定义为:一个或者多个通过许多“+”或者“-”操作符分隔而成的项。这些项自身可以定义为:由“*”或者“/”操作符分

隔而成的一个或者多个因子(factor)。通过把表达式分解成项,再把项分解成因子,就可以确保以正确的顺序来使用这些操作符了。

例如,"2*C5+D6"就是一个表达式,它由作为第一项的"2*C5"和作为第二项的"D6"构成。项"2*C5"是由作为第一个因子的"2"和作为第二个因子的"C5"组成的,而项"D6"则由一个单一的因子"D6"组成。一个因子可以是一个数("2")、一个单元格位置("C5"),或者是一个在圆括号内的表达式,在它们的前面可以有负号。

在图4.10中,定义了电子制表软件表达式的语法。对于语法(表达式、项和因子)中的每一个符号,都对应一个解析它的成员函数,并且函数的结构严格遵循语法。通过这种方式写出的解析器称为递归渐降解析器(recursive-descent parser)。

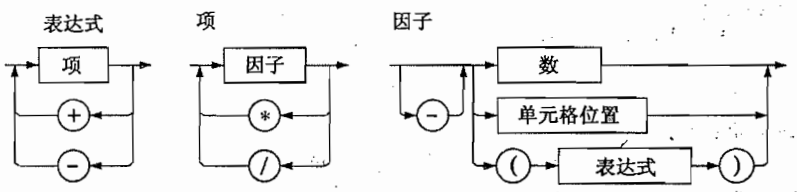


图4.10　用于电子制表软件表达式中的语法图

让我们先从evalExpression()开始,这个函数可以解析一个表达式:

```
QVariant Cell::evalExpression(const QString &str, int &pos) const
{
    QVariant result = evalTerm(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '+' && op != '-')
            return result;
        ++pos;

        QVariant term = evalTerm(str, pos);
        if (result.type() == QVariant::Double
                && term.type() == QVariant::Double) {
            if (op == '+') {
                result = result.toDouble() + term.toDouble();
            } else {
                result = result.toDouble() - term.toDouble();
            }
        } else {
            result = Invalid;
        }
    }
    return result;
}
```

首先,调用evalTerm()得到第一项的值。如果它后面紧跟的字符是"+"或者"-",那么就继续第二次调用evalTerm();否则,表达式就只包含一个单一项,并且把它的值作为整个表达式的值而返回。在得到前两项的值之后,根据操作符计算出这一操作的结果。如果两项都求出一个double值,就把计算出的结果当作一个double值;否则,把结果设置为Invalid。

像前面那样继续操作,直到再没有更多的项为止。这样做可以正确地进行,因为加法和减法都是左相关(left-associative)的;也就是说,"1-2-3"的意思是"(1-2)-3",而不是"1-(2-3)"。

```
QVariant Cell::evalTerm(const QString &str, int &pos) const
{
    QVariant result = evalFactor(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '*' && op != '/')
```

```
            return result;
        ++pos;

        QVariant factor = evalFactor(str, pos);
        if (result.type() == QVariant::Double
                && factor.type() == QVariant::Double) {
            if (op == '*') {
                result = result.toDouble() * factor.toDouble();
            } else {
                if (factor.toDouble() == 0.0) {
                    result = Invalid;
                } else {
                    result = result.toDouble() / factor.toDouble();
                }
            }
        } else {
            result = Invalid;
        }
    }
    return result;
}
```

除了 evalTerm()函数是处理乘法和除法这一点不同之外,它和 evalExpression()都很相似。在 evalTerm()中唯一的不同就是必须要避免除零,因为在一些处理器中这将是一个错误。尽管测试浮点数值是否相等通常并不明智,因为其中存在取舍问题,但是在这个防止除零的问题上,这样做相等性测试已经足够了。

```
QVariant Cell::evalFactor(const QString &str, int &pos) const
{
    QVariant result;
    bool negative = false;

    if (str[pos] == '-') {
        negative = true;
        ++pos;
    }

    if (str[pos] == '(') {
        ++pos;
        result = evalExpression(str, pos);
        if (str[pos] != ')')
            result = Invalid;
        ++pos;
    } else {
        QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
        QString token;

        while (str[pos].isLetterOrNumber() || str[pos] == '.') {
            token += str[pos];
            ++pos;
        }

        if (regExp.exactMatch(token)) {
            int column = token[0].toUpper().unicode() - 'A';
            int row = token.mid(1).toInt() - 1;

            Cell *c = static_cast<Cell *>(
                          tableWidget()->item(row, column));
            if (c) {
                result = c->value();
            } else {
                result = 0.0;
            }
        } else {
            bool ok;
            result = token.toDouble(&ok);
            if (!ok)
                result = Invalid;
```

```
        }
    }

    if (negative) {
        if (result.type() == QVariant::Double) {
            result = -result.toDouble();
        } else {
            result = Invalid;
        }
    }
    return result;
}
```

evalFactor()函数比 evalExpression()和 evalTerm()函数都要复杂一些。它先从计算因子是否为负开始。然后,判断它是否是从左圆括号开始的。如果是,就先把圆括号内的内容作为表达式并通过调用 evalExpression()来处理它。当解析到带圆括号的表达式时,evalExpression()调用 evalTerm(),evalTerm()调用 evalFactor(),evalFactor()则会再次调用 evalExpression()。这就是在解析器中出现递归调用的地方。

如果该因子不是一个嵌套表达式,就提取下一个记号,它应当是一个单元格的位置,或者也可能是一个数字。如果这个记号匹配 QRegExp,就把它认为是一个单元格引用并且对给定位置处的单元格调用 value()。该单元格可能在电子制表软件中的任何一个地方,并且它可能会依赖于其他的单元格。这种依赖不是什么问题,它们只会简单地触发更多的 value()调用和(对于那些"dirty"单元格)更多的解析处理,直到所有相关的单元格的值都得到计算为止。如果记号不是一个单元格的位置,那么就把它看作是一个数字。

如果单元格 A1 包含公式"=A1"时会发生什么呢?或者如果单元格 A1 包含公式"=A2"并且单元格 A2 包含公式"=A1"时又会发生什么呢?尽管还没有编写任何特定代码来检测这种循环依赖关系,但解析器可以通过返回一个无效的 QVariant 来完美地处理这一情况。之所以可以正常工作,是因为在调用 evalExpression()之前,我们会在 value()中把 cacheIsDirty 设置为 false,把 cachedValue设置为 Invalid。如果 evalExpression()对同一个单元格循环调用 value(),它就会立即返回 Invalid,并且这样就会使整个表达式等于 Invalid。

现在,我们已经完成了公式解析器。通过扩展因子的语法定义,公式解析器可以非常方便地处理那些在电子制表软件中像"sum()"和"avg()"一样的某些预定义函数。而另外一种比较容易的扩展方式是将"+"操作符实现为字符串连接(就像串联一样),这样就无需再对这个语法进行修改了。

第5章　创建自定义窗口部件

这一章讲解如何使用 Qt 开发自定义窗口部件。通过对一个已经存在的 Qt 窗口部件进行子类化或者直接对 QWidget 进行子类化，就可以创建自定义窗口部件。本章将示范这两种方式，并且也会说明如何把自定义窗口部件集成到 Qt 设计师中，这样就可以像使用内置的 Qt 窗口部件一样来使用它们了。最后，将通过展示一个使用了双缓冲技术（一种用于快速绘制的强大技术）的自定义窗口部件来结束这一章的内容。

5.1　自定义 Qt 窗口部件

在某些情况下，我们发现 Qt 窗口部件需要更多的自定义定制，这些定制可能要比它在 Qt 设计师里可设置的属性或者对它调用的那些函数更多一些。一个简单而直接的解决方法就是对相关的窗口部件类进行子类化并且使它能够满足我们的需要。

这一节将开发一个如图 5.1 所示的十六进制微调框，以说明是如何完成这一工作的。QSpinBox 一般只支持十进制整数，但是通过子类化方法，可以让它非常容易地接受并且显示十六进制数值。



图 5.1　HexSpinBox 窗口部件

```
#ifndef HEXSPINBOX_H
#define HEXSPINBOX_H

#include <QSpinBox>

class QRegExpValidator;

class HexSpinBox : public QSpinBox
{
    Q_OBJECT
public:
    HexSpinBox(QWidget *parent = 0);

protected:
    QValidator::State validate(QString &text, int &pos) const;
    int valueFromText(const QString &text) const;
    QString textFromValue(int value) const;

private:
    QRegExpValidator *validator;
};

#endif
```

HexSpinBox 从 QSpinBox 中继承了它的绝大多数功能，它提供了一个典型的构造函数，并且重新实现了 QSpinBox 中的三个虚函数。

```
#include <QtGui>

#include "hexspinbox.h"

HexSpinBox::HexSpinBox(QWidget *parent)
    : QSpinBox(parent)
{
    setRange(0, 255);
    validator = new QRegExpValidator(QRegExp("[0-9A-Fa-f]{1,8}"), this);
}
```

我们设置的默认范围是从 0 到 255(即从 0x00 到 0xFF)。对于十六进制微调框来说,这样的设置比 QSpinBox 的默认设置值范围(从 0 到 99)更为合适些。

用户可以通过单击微调框的向上和向下箭头或者在它的行编辑器中输入数值来修改微调框的当前值。在后一种情况中,我们想要严格控制用户输入的数据必须是合法的十六进制数字。为了实现这一点,我们使用一个 QRegExpValidator,它可以接受 1 到 8 个字符,所有这些字符都必须是集合{'0',…,'9','A',…,'F','a',…,'f'}中的一个。

```
QValidator::State HexSpinBox::validate(QString &text, int &pos) const
{
    return validator->validate(text, pos);
}
```

这个函数由 QSpinBox 调用,用来检查目前为止用户输入文本的合法性。有三种结果可能会出现:Invalid(无效,输入的文本与常规表达式不匹配)、Intermediate(部分有效部分无效,输入的文本是一个有效值中似是而非的一部分)以及 Acceptable(可以接受,输入的文本合法有效)。QRegExpValidator 有一个合适的 validate()函数,因此只需返回对其调用后的最终结果即可。理论上,应当为位于微调框范围之外的那些值返回 Invalid 或者 Intermediate,但是 QSpinBox 具有很好的自适应性,它可以在没有任何帮助的情况下检测出那种情况。

```
QString HexSpinBox::textFromValue(int value) const
{
    return QString::number(value, 16).toUpper();
}
```

textFromValue()函数把一个整数值转换成一个字符串。当用户按下微调框的向上或者向下箭头时,QSpinBox 会调用它来更新微调框的编辑器部分。我们使用静态函数 QString::number(),将其第二个参数设置为 16,把该值转换为小写格式的十六进制字符串,并且对结果调用 QString::toUpper(),使其成为大写格式的结果。

```
int HexSpinBox::valueFromText(const QString &text) const
{
    bool ok;
    return text.toInt(&ok, 16);
}
```

valueFromText()函数执行从字符串到整数值的逆向转换。当用户在微调框的编辑器部分输入一个值并且按下 Enter 时,QSpinBox 就会调用它。我们使用 QString::toInt()试着把当前文本转换成一个整数值,当然还是使用 16 作为基数。如果这个字符串不是有效的十六进制数,那么就把 ok 设置为 false,并且由 toInt()返回一个 0。在这里,不需要考虑这种可能出现的情况,因为此处的验证器只允许输入有效的十六进制字符串。我们希望能够给 toInt()的第一个参数传递的是一个空指针,而不是传递一个虚拟变量(ok)的地址。

现在已经完成了这个十六进制微调框。自定义其他 Qt 窗口部件也可以遵循相同的模式:选择一个合适的 Qt 窗口部件,对它进行子类化,并且通过重新实现一些虚函数来改变它的行为即可。如果我们想做的全部就是对一个已经存在的窗口部件的外观进行自定义设置,那么只需对其应用一个样式表或者重新实现一种自定义风格即可,而不必对其进行子类化,就像第 19 章中说明的那样。

5.2　子类化 QWidget

许多自定义窗口部件都是对现有窗口部件的简单组合,不论它们是内置的 Qt 窗口部件,还是

其他一些像 HexSpinBox 这样的自定义窗口部件。通过对现有窗口部件的组合构建而成的自定义窗口部件通常都可以在 Qt 设计师中开发出来:

- 使用"Widget"模板创建一个新窗体。
- 把一些必需的窗口部件添加到这个窗体上,并且对它们进行摆放。
- 设置一些信号和槽的连接。
- 如果通过信号和槽不能获得所需的行为,则只需在类中添加一些必要的代码即可——这个类需要同时从 QWidget 类和 uic 生成的类中派生出来。

当然,要对这些现有窗口部件进行组合,也完全可以通过手写代码方式来加以实现。但无论使用的是哪种方式,最终生成的类都会是 QWidget 类的一个子类。

如果窗口部件本身没有任何信号和槽,并且它也没有重新实现任何虚函数,那么我们甚至还是有可能通过对现有窗口部件的组合而不是通过子类化的方式来生成这样的窗口部件。这就是在第 1 章创建 Age 应用程序时所使用的方法,其中用到了一个 QWidget、一个 QSpinBox 以及一个 QSlider。虽然如此,也还是可以很容易地通过子类化 QWidget,并且在它的子类构造函数中创建 QSpinBox 和 QSlider 的方式来做到这一点。

当手里没有任何一个 Qt 窗口部件能够满足任务要求,并且也没有办法通过组合现有窗口部件来满足所需的期望结果时,仍旧可以创建出我们想要的窗口部件来。要实现这一点,只需通过子类化 QWidget,并且通过重新实现一些用来绘制窗口部件和响应鼠标点击的事件处理器即可。这一方法给了我们定义并且控制自己的窗口部件的外观和行为的完全自由。Qt 的一些内置窗口部件,像 QLabel、QPushButton 和 QTableWidget,都是通过这种方法得以重新实现的。如果它们没有在 Qt 中存在,那么还是完全有可能以与平台无关的方式使用 QWidget 所提供的公有函数来创建它们。

为了说明如何使用这种方法编写一个自定义窗口部件,我们将会创建一个如图 5.2 所示的 IconEditor 窗口部件。这个 IconEditor 本来是一个用于图标编辑器程序中的窗口部件。

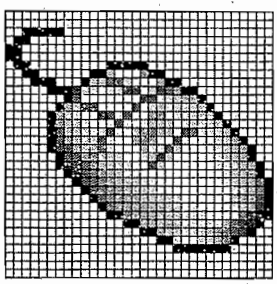


图 5.2 IconEditor 窗口部件

实际上,在我们开始潜心研究和创建一个自定义窗口部件之前,还是很有必要先去检查一下是否已经有了可用的相关窗口部件,无论是在 Qt Solution(http://www.trolltech.com/products/qt/addon/solutions/catalog/4/)中还是在商业或者非商业第三方(http://www.trolltech.com/products/qt/3rdparty/)那里都行,因为这样将很有可能会节省许多时间和精力。在本例中,假设没有可用的合适窗口部件,因而需要创建我们自己的窗口部件。

让我们先从它的头文件开始看起:

```
#ifndef ICONEDITOR_H
#define ICONEDITOR_H

#include <QColor>
#include <QImage>
#include <QWidget>

class IconEditor : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)
```

```
public:
    IconEditor(QWidget *parent = 0);

    void setPenColor(const QColor &newColor);
    QColor penColor() const { return curColor; }
    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom; }
    void setIconImage(const QImage &newImage);
    QImage iconImage() const { return image; }
    QSize sizeHint() const;
```

类 IconEditor 使用 Q_PROPERTY()宏声明了三个自定义属性:penColor、iconImage 和 zoomFactor。每一个属性都有一个数据类型、一个"读"函数和一个作为可选项的"写"函数。例如,penColor 属性的类型是 QColor,并且可以使用 penColor()和 setPenColor()函数对它进行读写。

当我们在 Qt 设计师中使用这个窗口部件时,在 Qt 设计师属性编辑器里,那些继承于 QWidget 的属性下面,将会显示这些自定义属性。这些属性可以是由 QVariant 所支持的任何类型。对于定义属性的类,Q_OBJECT宏是必需的。

```
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);

private:
    void setImagePixel(const QPoint &pos, bool opaque);
    QRect pixelRect(int i, int j) const;

    QColor curColor;
    QImage image;
    int zoom;
};

#endif
```

IconEditor 重新实现了 QWidget 中的三个保护函数,并且也拥有一些自己的私有函数和私有变量。这三个私有变量保存这三个属性的值。

实现文件是从 IconEditor 的构造函数开始的。

```
#include <QtGui>

#include "iconeditor.h"

IconEditor::IconEditor(QWidget *parent)
    : QWidget(parent)

{
    setAttribute(Qt::WA_StaticContents);
    setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);

    curColor = Qt::black;
    zoom = 8;

    image = QImage(16, 16, QImage::Format_ARGB32);
    image.fill(qRgba(0, 0, 0, 0));
}
```

构造函数有一些巧妙的地方,比如这里的 Qt::WA_StaticContents 属性和 setSizePolicy()调用。我们将简单地讨论一下它们。

画笔的颜色被设置为黑色。缩放因子(zoom factor)被设置为 8,也就是说,图标中的每一个像素都将会显示成一个 8×8 的正方形。

图标数据会保存在 image 成员变量中,并且可以通过 setIconImage()和 iconImage()函数对它们进行访问。当用户打开一个图标文件时,图标编辑器程序通常会调用 setIconImage()函数;当用户

想保存这个图标时,它就会调用 iconImage()来重新得到这个图标。Image 变量的类型是 QImage。我们把它初始化为 16×16 的像素大小和 32 位的 ARGB 颜色格式,这种颜色格式可以支持半透明效果。通过填充一种透明的颜色,就可以清空 image 中的数据。

QImage 类使用一种与硬件无关的方式来存储图像。可以把它设置成使用 1 位、8 位或者 32 位色深。一个具有 32 位色深的图像分别对每一个像素各使用 8 位来存储它的红、绿、蓝分量。剩余的 8 位存储这个像素的 alpha 分量(即不透明度)。例如,一个纯红颜色的红、绿、蓝和 alpha 分量的值分别是 255、0、0 和 255。在 Qt 中,这种颜色可以通过如下形式给定:

```
QRgb red = qRgba(255, 0, 0, 255);
```

或者,由于该颜色是不透明的,所以可以表示为:

```
QRgb red = qRgb(255, 0, 0);
```

QRgb 只是 unsigned int 类型的一个 typedef(类型别名),并且 qRgb()和 qRgba()都是用来把它们的参数组合成一个 32 位 ARGB 整数值的内联函数。也可能写成这样的形式:

```
QRgb red = 0xFFFF0000;
```

这里的第 1 个 FF 对应于 alpha 分量,第 2 个 FF 对应于红色分量。在 IconEditor 的构造函数中,我们通过使用 0 作为 alpha 分量而构成的透明色来填充这个 QImage。

Qt 提供了两种存储颜色的类型:QRgb 和 QColor。虽然 QRgb 仅仅是一个用在 QImage 中存储 32 位像素数据的类型别名,但 QColor 则是一个具有许多有用函数并且在 Qt 中广泛用于存储颜色的类。在 QIconEditor 窗口部件中,只有在处理 QImage 时,我们才使用 QRgb,而对于其他任意东西,包括这里的 penColor 属性,我们都只使用 QColor。

```
QSize IconEditor::sizeHint() const
{
    QSize size = zoom * image.size();
    if (zoom >= 3)
        size += QSize(1, 1);
    return size;
}
```

sizeHint()函数是从 QWidget 中重新实现的,并且可以返回一个窗口部件的理想大小。在这里,我们用缩放因子乘以图像的尺寸大小作为这个窗口部件的理想大小,但如果缩放因子是 3 或者更大,那么在每一个方向上需要再额外增加一个像素,以便可以容纳一个网格线。(如果缩放因子是 2 或者 1,就不必再显示网格线,因为这些网格线将几乎不能再给图标的像素留下任何空间。)

在和布局联合使用时,窗口部件的大小提示非常有用。当 Qt 的布局管理器摆放一个窗体的子窗口部件时,它会尽可能多地考虑这些窗口部件的大小提示。为了能够让 IconEditor 成为一个具有良好布局的窗口部件,它必须报告一个可靠的大小提示。

除了大小提示,窗口部件还有一个大小策略,它会告诉布局系统是否可以对这个窗口部件进行拉长或者缩短。通过在构造函数中调用以 QSizePolicy::Minimum 为水平和垂直大小策略的 setSizePolicy(),会告诉负责管理这个窗口部件的任意布局管理器,这个窗口部件的大小提示就是它的最小尺寸大小。换句话说,如果需要的话,可以拉长这个窗口部件,但是决不允许把它缩短到比它的大小提示还要小的尺寸。在 Qt 设计师中,通过设置这个窗口部件的 sizePolicy 属性也可以实现这一特性。第 6 章将会解释各式各样的大小策略。

```
void IconEditor::setPenColor(const QColor &newColor)
{
    curColor = newColor;
}
```

函数 setPenColor()会设置画笔的当前颜色。这个颜色将会用于此后新绘制的像素中。

```
void IconEditor::setIconImage(const QImage &newImage)
{
    if (newImage != image) {
        image = newImage.convertToFormat(QImage::Format_ARGB32);
        update();
        updateGeometry();
    }
}
```

函数 setIconImage()会设置需要编辑的图像。如果这个图像还不是我们正在编辑的图像，则会调用 convertToFormat()把它变成一个带 alpha 缓冲的 32 位图像。在其他代码中，将假设图像数据是存储在 32 位的 ARGB 值中的。

在设置完 image 变量后，调用 QWidget::update()，它会使用新的图像强制重绘这个窗口部件。接下来，调用 QWidget::updateGeometry()，告诉包含这个窗口部件的任意布局，这个窗口部件的大小提示已经发生改变了。于是，该布局将会自动适应这个新的大小提示。

```
void IconEditor::setZoomFactor(int newZoom)
{
    if (newZoom < 1)
        newZoom = 1;

    if (newZoom != zoom) {
        zoom = newZoom;
        update();
        updateGeometry();
    }
}
```

setZoomFactor()函数设置图像的缩放因子。为了避免在其他地方被 0 除，应纠正任何小于 1 的值。之后，会再次调用 update()和 updateGeometry()来重新绘制该窗口部件，以便可以把大小提示的变化通知给其他任何一个负责管理它的布局。

在头文件中，我们把 penColor()、iconImage()和 zoomFactor()函数都实现成了内联函数。

现在查看 paintEvent()函数的代码。这个函数是 IconEditor 最为重要的函数。只要需要重新绘制窗口部件，就会调用它。它在 QWidget 中的默认实现什么都不做，这样就留下了一个空白的窗口部件。

就像在第 3 章中碰到的 closeEvent()函数一样，paintEvent()也是一个事件处理器。Qt 还有很多其他的事件处理器，每一个都对应一种不同类型的事件。第 7 章将会进一步深入介绍事件的处理。

当产生一个绘制事件并且调用 paintEvent()函数的时候，会出现如下几种情况：

- 在窗口部件第一次显示时，系统会自动产生一个绘制事件，从而强制绘制这个窗口部件本身。
- 当重新调整窗口部件大小的时候，系统也会产生一个绘制事件。
- 当窗口部件被其他窗口部件遮挡，然后又再次显示出来的时候，就会对那些隐藏的区域产生一个绘制事件(除非这个窗口系统存储了整个区域)。

也可以通过调用 QWidget::update()或者 QWidget::repaint()来强制产生一个绘制事件。这两个函数之间的区别是：repaint()函数会强制产生一个即时的重绘事件，而 update()函数则只是在 Qt 下一次处理事件时才简单地调用一个绘制事件。(如果窗口部件在屏幕上是不可见的，那么这两个函数会什么也不做。)如果多次调用 update()，Qt 就会把连续多次的绘制事件压缩成一个单一的绘制事件，这样就可以避免闪烁现象。在 IconEditor 中，我们总是使用 update()函数。

这里给出的是 paintEvent()函数的代码:

```
void IconEditor::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    if (zoom >= 3) {
        painter.setPen(palette().foreground().color());
        for (int i = 0; i <= image.width(); ++i)
            painter.drawLine(zoom * i, 0,
                             zoom * i, zoom * image.height());
        for (int j = 0; j <= image.height(); ++j)
            painter.drawLine(0, zoom * j,
                             zoom * image.width(), zoom * j);
    }

    for (int i = 0; i < image.width(); ++i) {
        for (int j = 0; j < image.height(); ++j) {
            QRect rect = pixelRect(i, j);
            if (!event->region().intersect(rect).isEmpty()) {
                QColor color = QColor::fromRgba(image.pixel(i, j));
                if (color.alpha() < 255)
                    painter.fillRect(rect, Qt::white);
                painter.fillRect(rect, color);
            }
        }
    }
}
```

首先,在窗口部件上构建了一个 QPainter 对象。如果缩放因子是 3 或者比 3 还要大,就使用 QPainter::drawLine()函数绘制构成网格的水平线段和垂直线段。

对于 QPainter::drawLine()的调用遵循这样的语法:

```
painter.drawLine(x1, y1, x2, y2);
```

其中,$(x1,y1)$是线段的一个端点的位置,$(x2,y2)$是线段的另一个端点的位置。这个函数还有另外一个重载版本的形式,它使用两个 QPoint 来代替这里给出的 4 个 int 值。

Qt 窗口部件的左上角处的位置坐标是(0,0),右下角的位置坐标是(width()-1,height()-1)。这和常用的笛卡儿坐标系(Cartesian coordinate system)相似,只不过它的 y 坐标是自上而下的,如图 5.3 所示。通过使用平移、缩放、旋转和错切等变换,可以改变 QPainter 的坐标系。第 8 章将会讲述这些内容。

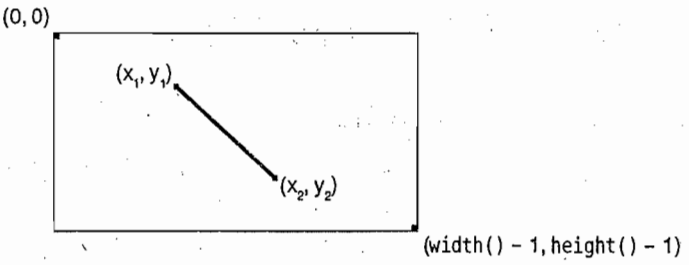


图 5.3 使用 QPainter 绘制一条线段

在对 QPainter 调用 drawLine()之前,使用了 setPen()设置线段的颜色。本来也可以直接通过代码来指定像黑色或者灰色这样的颜色,但是使用窗口部件的调色板(palette)会更好些。

每一个窗口部件都会配备一个调色板,由它来确定做什么事应该使用什么颜色。例如,对于窗口部件的背景色会有一个对应的调色板条目(通常是亮灰色),并且对于文本的背景色也会对应一个调色板条目(通常是黑色)。默认情况下,一个窗口部件的调色板会采用窗口系统的颜色主题。通过使用调色板中的这些颜色,可以确保 IconEditor 能够尊重用户的选择。

一个窗口部件的调色板由三个颜色组构成：激活组（Active）、非激活组（Inactive）和不可用组（Disabled）。应该使用哪一个颜色组取决于该窗口部件的当前状态：

- Active 颜色组可用于当前激活窗口中的那些窗口部件。
- Inactive 颜色组可用于其他窗口中的那些窗口部件。
- Disabled 颜色组可用于任意窗口中的那些不可用窗口部件。

QWidget::palette()函数可以返回窗口部件的调色板，它是一个 QPalette 型对象。颜色组给定为 QPalette::ColorGroup 型枚举变量值。

如果我们希望获得一个用于绘制的适当的画笔或者颜色，正确的方法就是使用当前调色板。它可以通过 QWidget::palette()而获得，并且也可以使用所需的角色，例如，QPalette::foreground()。每个角色函数都可以返回一个画笔，它通常就正是我们所想要的东西，但如果我们只需要颜色的话，则可以将其从画笔中提取出来，就像 paintEvent()中所做的那样。默认情况下，返回的那些画笔都是能够适用于窗口部件的状态的，因而就没有必要再去给定颜色组。

以图像自身的绘制作为 paintEvent()函数的结尾。对 IconEditor::pixelRect()的调用会返回一个 QRect，其中定义了需要重新绘制的区域。（图 5.4 说明了矩形是如何绘制的。）作为简单的优化处理方法，我们没有对落在这个区域之外的像素进行重新绘制。

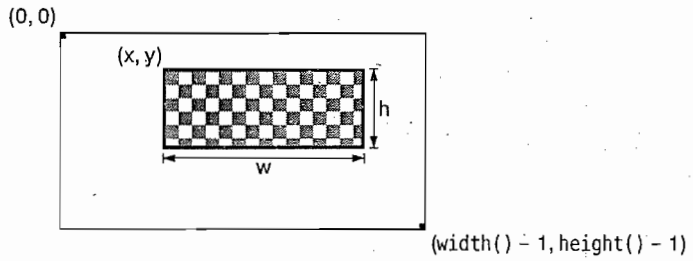


图 5.4　使用 QPainter 绘制一个矩形

我们调用 QPainter::fillRect()来绘制一个缩放后的像素。QPainter::fillRect()带一个 QRect 和一个 QBrush。通过传递一个用作画笔的 QColor，我们获得了一个实心填充图案。如果该颜色并非完全不透明（它的 alpha 通道小于 255），就会先绘制出一个白色的背景来。

```
QRect IconEditor::pixelRect(int i, int j) const
{
    if (zoom >= 3) {
        return QRect(zoom * i + 1, zoom * j + 1, zoom - 1, zoom - 1);
    } else {
        return QRect(zoom * i, zoom * j, zoom, zoom);
    }
}
```

pixelRect()函数返回一个适用于 QPainter::fillRect()的 QRect。这里的参数 i 和 j 是 QImage 的像素坐标，而不是窗口部件中的坐标。如果缩放因子是 1，那么这两个坐标系就可以恰好一致了。

QRect 构造函数具有 QRect(x, y, *width*, *height*)的语法形式，这里的(x, y)是这个矩形左上角的位置坐标，而 *width* × *height* 就是矩形的尺寸大小。如果缩放因子是 3 或者更大，则可以在矩形的水平和竖直方向大小上都减去一个像素，以便在填充时不会覆盖那些网格线。

```
void IconEditor::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->button() == Qt::RightButton) {
```

```
        setImagePixel(event->pos(), false);
    }
}
```

当用户按下鼠标按钮时,系统就会产生一个“鼠标按下”事件。通过重新实现QWidget::mousePressEvent(),就可以响应这一事件,并且可以对鼠标光标下的图像像素进行设置或者清空。

如果用户按下了鼠标左键,则使用 true 作为调用私有函数 setImagePixel()的第二个参数,告诉它要把这个像素设置成当前画笔的颜色。如果用户按下了鼠标右键,也会调用setImagePixel(),但这一次是通过传递 false 来清空这个像素。

```
void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->buttons() & Qt::RightButton) {
        setImagePixel(event->pos(), false);
    }
}
```

mouseMoveEvent()处理“鼠标移动”事件。默认情况下,只有当用户按住一个键不放的时候,才会产生这些事件。通过调用 QWidget::setMouseTracking()则有可能改变这一行为,但是在这个例子中不需要这样做。

就像按下鼠标左键或者右键可以设置或者清空一个像素一样,把按键按下不放并且悬停在另一个像素上也足可以设置或者清空一个像素。由于有可能会同时按下多个键,所以最终结果实际是 QMouseEvent::buttons()的返回值与鼠标的按键按照按位“或”(OR)运算之后的结果。可以使用“&”操作符来测试某个特定键是否按下了,并且如果是这样的话,就调用 setImagePixel()。

```
void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
{
    int i = pos.x() / zoom;
    int j = pos.y() / zoom;

    if (image.rect().contains(i, j)) {
        if (opaque) {
            image.setPixel(i, j, penColor().rgba());
        } else {
            image.setPixel(i, j, qRgba(0, 0, 0, 0));
        }

        update(pixelRect(i, j));
    }
}
```

setImagePixel()函数是从 mousePressEvent()和 mouseMoveEvent()中得到调用的,用来设置或者清空一个像素。pos 参数是鼠标在窗口部件中的位置。

第一步是把鼠标的位置从窗口部件的坐标转换到图像的坐标。这可以通过使用鼠标的 x()和 y()分量除以缩放因子完成。接下来,检查该点是否位于正确的范围之内。使用QImage::rect()和QRect::contains()可以很容易地完成这一检查过程。这样就可以高效地检查出 i 是不是在 0 和 image.width()-1 之间,j 是否位于 0 和image.height()-1之间。

根据 opaque 参数,我们可以设置或者清空图像中的像素。清空一个像素,实际就是把它设置成透明。我们必须把画笔的 QColor 转换为一个用于调用 QImage::setPixel()的 32 位 ARGB 值。最后,对需要重新绘制的区域调用带 QRect 的 update()。

现在已经查看了各个成员函数,下面将回到构造函数中使用的 Qt::WA_StaticContents属性上。这个属性告诉 Qt,当重新改变窗口部件的大小时,这个窗口部件的内容并没有发生变化,而且它的

内容仍旧保留从窗口部件左上角开始的特性。当重新定义窗口部件的大小时，通过使用这个信息，Qt 就可以避免对已经显示区域的重新绘制。图 5.5 图示了这一情形。



图 5.5　重新改变 Qt::WA_StaticContents 窗口部件的大小

通常情况下，当重新定义一个窗口部件的大小时，Qt 会为窗口部件的整个可见区域生成一个绘制事件。但是如果该窗口部件在创建时使用了 Qt::WA_StaticContents 属性，那么绘制事件的区域就会被严格限定在之前没有被显示的像素部分上。这也就意味着，如果重新把窗口部件改变为比原来还要小的尺寸，那么就根本不会产生任何绘制事件。

IconEditor 窗口部件现在就完成了。通过使用前几章中的知识和例子，我们编写代码，把 IconEditor 作为一个独立的窗口，或者作为 QMainWindow 中的一个中央窗口部件、布局中的一个子窗口部件以及 QScrollArea(参见 6.4 节)中的一个子窗口部件。在下一节中，我们将会看到如何把它集成到 Qt 设计师中。

5.3　在 Qt 设计师中集成自定义窗口部件

在 Qt 设计师中使用自定义窗口部件之前，我们必须让 Qt 设计师先察觉到它们的存在。有两种方法可以完成这一任务：改进法(promotion)和插件法(plugin)。

改进法是最为快捷和简单的方法。这种方法包括：选择一个内置的 Qt 窗口部件，但该窗口部件要和我们自定义的窗口部件具有相类似的应用程序编程接口，并在 Qt 设计师的自定义窗口部件对话框(如图 5.6 所示)中填写一些与这个窗口部件相关的信息。然后，这个自定义窗口部件就可用于由 Qt 设计师开发的窗体中，尽管在编辑或者预览该窗体时它有可能仍旧显示为与之相关的内置 Qt 窗口部件的形式。

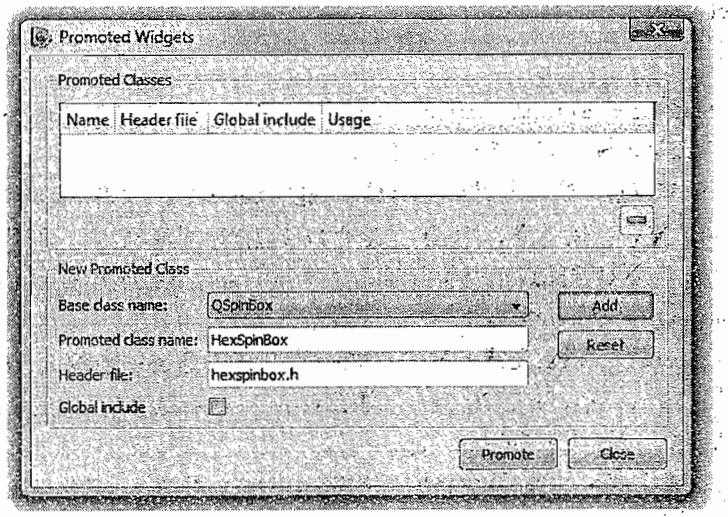


图 5.6　Qt 设计师中的自定义窗口部件对话框

以下给出了如何使用这种方法在窗体中插入一个 HexSpinBox 窗口部件的步骤:

1. 从 Qt 设计师的窗口部件工具盒中拖动一个 QSpinBox 到窗体中,由此创建一个 QSpinBox。
2. 右键单击这个微调框,并且从上下文菜单中选择"Promote to Custom Widget"(改进成自定义窗口部件)。
3. 在弹出的对话框中,填入"HexSpinBox"作为类的名字,填入"hexspinbox.h"作为头文件的名字。

这三步就足够了。由 uic 生成的代码将会包含 hexspinbox.h,而不是 <QSpinBox>,并且会生成一个 HexSpinBox 的实例。在 Qt 设计师中,将会用 QSpinBox 的图标来代表 HexSpinBox 窗口部件,从而允许我们设置 QSpinBox 的所有属性(例如,它的作用范围和当前值)。

改进法的缺点是:在 Qt 设计师中,无法对自定义窗口部件中的那些特定属性进行访问,并且也无法对这个窗口部件自身进行绘制。所有这两个问题都可以通过使用插件法得到解决。

插件法需要创建一个插件库,Qt 设计师可以在运行时加载这个库,并且可以利用该库创建窗口部件的实例。在对窗体进行编辑或者用于窗体预览时,Qt 设计师就会用到这个真正的窗口部件,这要归功于 Qt 的元对象系统,Qt 设计师才可以动态获取它的这些属性的列表。为了说明它是如何工作的,我们将把前一小节中的 IconEditor 集成为一个插件。

首先,必须对 QDesignerCustomWidgetInterface 进行子类化,并且需要重新实现一些虚函数。假定插件的源代码放在一个名为 iconeditorplugin 的目录中,并且 IconEditor 的源代码放在与 iconeditorplugin 目录同级的 iconeditor 目录中。

以下是类的定义:

```
#include <QDesignerCustomWidgetInterface>

class IconEditorPlugin : public QObject,
                         public QDesignerCustomWidgetInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)

public:
    IconEditorPlugin(QObject *parent = 0);

    QString name() const;
    QString includeFile() const;
    QString group() const;
    QIcon icon() const;
    QString toolTip() const;
    QString whatsThis() const;
    bool isContainer() const;
    QWidget *createWidget(QWidget *parent);
};
```

IconEditorPlugin 子类是一个封装了这个 IconEditor 窗口部件的工厂(factory)类。它从 QObject 和 QDesignerCustomWidgetIterface 中派生而来,并且使用 Q_INTERFACES() 宏通知 moc:这里的第二个基类是一个插件接口。Qt 设计师会使用这些函数创建这个类的各个实例并且获取相关信息。

```
IconEditorPlugin::IconEditorPlugin(QObject *parent)
    : QObject(parent)
{
}
```

构造函数非常简单。

```
QString IconEditorPlugin::name() const
{
    return "IconEditor";
}
```

name()函数返回由该插件提供的这个窗口部件的名字。

```
QString IconEditorPlugin::includeFile() const
{
    return "iconeditor.h";
}
```

includeFile()函数返回由该插件封装的特定窗口部件的头文件的名称。这个头文件会包含在由 uic 工具所生成的代码中。

```
QString IconEditorPlugin::group() const
{
    return tr("Image Manipulation Widgets");
}
```

group()函数返回自定义窗口部件所应属于的窗口部件工具箱群组的名字。但如果以这个名字命名的窗口部件工具箱群组还没有使用到,那么 Qt 设计师将会为这个窗口部件创建一个新群组。

```
QIcon IconEditorPlugin::icon() const
{
    return QIcon(":/images/iconeditor.png");
}
```

icon()函数返回一个图标,可以在 Qt 设计师窗口部件工具箱中用它来代表自定义窗口部件。这里,我们假定 IconEditorPlugin 有一个与之相关联的 Qt 资源文件(resource file),其中有适当的一项可作为图标编辑器图像。

```
QString IconEditorPlugin::toolTip() const
{
    return tr("An icon editor widget");
}
```

toolTip()函数返回一个工具提示信息,可在鼠标悬停在 Qt 设计师窗口部件工具箱中的自定义窗口部件上时显示该信息。

```
QString IconEditorPlugin::whatsThis() const
{
    return tr("This widget is presented in Chapter 5 of <i>C++ GUI "
              "Programming with Qt 4</i> as an example of a custom Qt "
              "widget.");
}
```

whatsThis()函数返回用于 Qt 设计师“What's This?”中显示的文本。

```
bool IconEditorPlugin::isContainer() const
{
    return false;
}
```

如果该窗口部件还可以包含其他窗口部件,isContainer()函数就返回 true;否则,它就返回 false。例如,QFrame 就是一个可以包含其他窗口部件的窗口部件。一般情况下,任何窗口部件都可以再包含其他窗口部件,但是当 isContainer()返回 false 时,Qt 设计师则不会允许它再包含其他的窗口部件。

```
QWidget *IconEditorPlugin::createWidget(QWidget *parent)
{
    return new IconEditor(parent);
}
```

Qt 设计师会调用 createWidget()函数,利用给定的父对象创建该窗口部件类的一个实例。

```
Q_EXPORT_PLUGIN2(iconeditorplugin, IconEditorPlugin)
```

在实现该插件类的源文件的末尾,必须使用Q_EXPORT_PLUGIN2()宏,从而可以在Qt设计师中使用这个插件。第一个参数是希望给插件的名字,第二个参数是实现该插件类的名字。

用于构建该插件的.pro文件看起来如下所示:

```
TEMPLATE      = lib
CONFIG       += designer plugin release
HEADERS       = ../iconeditor/iconeditor.h \
                iconeditorplugin.h
SOURCES       = ../iconeditor/iconeditor.cpp \
                iconeditorplugin.cpp
RESOURCES     = iconeditorplugin.qrc
DESTDIR       = $$[QT_INSTALL_PLUGINS]/designer
```

qmake构建工具已经构建了一些预定义变量。$$[QT_INSTALL_PLUGINS]就是它们当中的一个,它保存了指向Qt安装目录中plugins目录所在的路径。当键入make或者nmake来构建该插件时,它就会自动把自己安装到Qt设计师的plugins/designer目录中。插件一旦构建完毕,在Qt设计师中就可以像使用其他内置的Qt窗口部件一样来使用IconEditor窗口部件。

如果想在Qt设计师中集成多个自定义窗口部件,则既可以为每个窗口部件创建一个插件,也可以通过从QDesignerCustomWidgetCollectionInterface中派生的方式把它们组合成一个单一插件。

5.4 双缓冲

双缓冲(double buffering)是一种图形用户界面编程技术,它包括把一个窗口部件渲染到一个脱屏像素映射(off-screen pixmap)中以及把这个像素映射复制到显示器上。在Qt的早期版本中,这种技术通常用于消除屏幕的闪烁以及为用户提供一个漂亮的用户界面。

在Qt 4中,QWidget会自动处理这些情况,所以我们很少需要考虑窗口部件的闪烁问题。尽管如此,但如果窗口部件的绘制非常复杂并且需要连续不断地重复绘制时,明确指定使用双缓冲则是非常有用的事情。于是就可以把这个窗口部件固定不变地存储成一个像素映射,这样就总可以为下一个绘制事件做好准备,并且一旦收到绘制事件,就可以把这个像素映射复制到窗口部件上。当我们想做一些小的修改时,比如一个橡皮筋选择框的绘制,此时并不需要对整个窗口部件进行重复绘制和计算,从而就会显得特别有用。

本章将通过回顾在图5.7和图5.9中显示的Plotter自定义窗口部件来结束。这个窗口部件使用了双缓冲技术,并且也对Qt编程中的一些其他方面的知识进行了示范,包括像键盘事件处理、手动布局和坐标系统等。

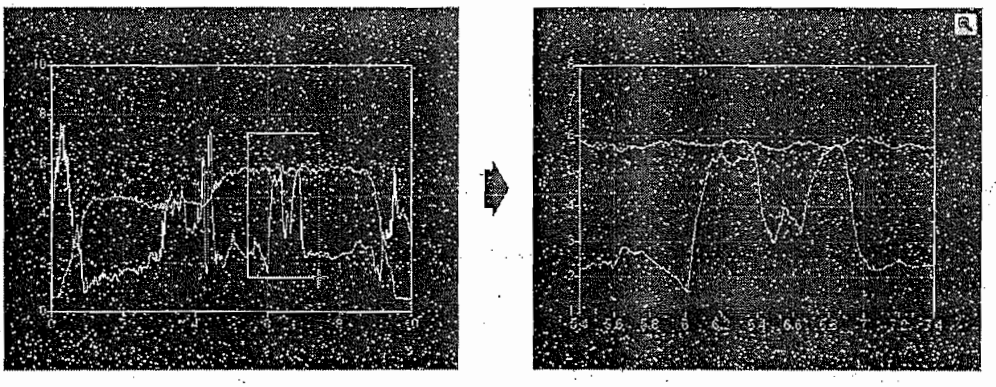


图5.7 Plotter窗口部件中的放大操作

对于需要具有一个图形处理或者图形测绘窗口部件的真正应用程序来说，最好还是使用那些可以获取的第三方窗口部件，而不是像这里所做的那样，去创建一个自定义窗口部件。例如，我们或许应当使用来自 http://www.ics.com/的 GraphPak，来自 http://www.kdab.net/的 KD Chart，或者是来自 http://qwt.sourceforge.net/的 Qwt。

Plotter 窗口部件可以按照给定的矢量坐标绘制一条或者多条曲线。用户可以在图像中拖拽一条橡皮筋选择框，并且 Plotter 将会对由这个橡皮筋选择框选定的区域进行放大。用户通过单击图上的一个点、按住鼠标左键并且把鼠标拖动到另外一个位置，就可以拖拽出一个橡皮筋选择框，然后就可以松开鼠标按键。Qt 为绘制橡皮筋选择框提供了类，但这里通过我们自己来绘制它，以提供更好的视觉控制效果，并且藉此说明双缓冲技术。

通过多次拖拽出橡皮筋选择框，可以重复进行放大操作。使用 Zoom Out 按钮可以进行缩小操作，而使用 Zoom In 按钮又可以再放大回来。Zoom In 和 Zoom Out 按钮在第一次出现的时候就处于启用状态。这样的话，如果用户没有对这个曲线图进行缩放，它们也不会弄乱图形的显示效果。

Plotter 窗口部件可以保存任意条曲线的数据。它还维护着一个 PlotSettings 堆栈对象，而这每一个堆栈对象都对应一个特定的缩放级别。

让我们来查看一下这个类，先从 plotter.h 开始：

```
#ifndef PLOTTER_H
#define PLOTTER_H

#include <QMap>
#include <QPixmap>
#include <QVector>
#include <QWidget>

class QToolButton;
class PlotSettings;

class Plotter : public QWidget
{
    Q_OBJECT

public:
    Plotter(QWidget *parent = 0);

    void setPlotSettings(const PlotSettings &settings);
    void setCurveData(int id, const QVector<QPointF> &data);
    void clearCurve(int id);
    QSize minimumSizeHint() const;
    QSize sizeHint() const;

public slots:
    void zoomIn();
    void zoomOut();
```

首先关注 plotter 头文件中包含的相应的 Qt 类头文件。在文件的开始部分，还前置声明了一些带指针或引用的类。

在 Plotter 类中，提供三个公有函数来用于创建绘图区(plot)，并且用两个公有槽来响应图形的放大和缩小操作。还重新实现了 QWidget 中的 minimumSizeHint()和 sizeHint()两个函数。我们把曲线的顶点存储为 QVector <QPointF> ，这里的 QPointF 是一个具有浮点数形式的 QPoint。

```
protected:
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
```

```
    void keyPressEvent(QKeyEvent *event);
    void wheelEvent(QWheelEvent *event);
```

在这个类的 protected 段中,声明了所有需要重新实现的 QWidget 事件处理器。

```
private:
    void updateRubberBandRegion();
    void refreshPixmap();
    void drawGrid(QPainter *painter);
    void drawCurves(QPainter *painter);

    enum { Margin = 50 };

    QToolButton *zoomInButton;
    QToolButton *zoomOutButton;
    QMap<int, QVector<QPointF> > curveMap;
    QVector<PlotSettings> zoomStack;
    int curZoom;
    bool rubberBandIsShown;
    QRect rubberBandRect;
    QPixmap pixmap;
};
```

在这个类的 private 段,声明了一些用于绘制这个窗口部件的函数、一个常量和几个成员变量。Margin 常量可以为图形区域的周围提供一些空间。

在这些成员变量中,pixmap 变量的类型为 QPixmap。这个变量对整个窗口部件的绘制数据的进行了复制保存,这和屏幕上显示的图形是相同的。绘图区总是先在脱屏像素映射上绘制图形。然后,才把这一像素映射复制到窗口部件中。

```
class PlotSettings
{
public:
    PlotSettings();

    void scroll(int dx, int dy);
    void adjust();
    double spanX() const { return maxX - minX; }
    double spanY() const { return maxY - minY; }

    double minX;
    double maxX;
    int numXTicks;
    double minY;
    double maxY;
    int numYTicks;

private:
    static void adjustAxis(double &min, double &max, int &numTicks);
};

#endif
```

PlotSettings 类给定了 x 轴和 y 轴的范围,以及在这些轴上刻度标记符的数量。图 5.8 给出了一个 PlotSettings 对象和一个 Plotter 窗口部件之间的对应关系。

依照惯例,把 numXTicks 和 numYTicks 都会减去 1。如果 numXTicks 是 5,那么 Plotter 实际会在 x 轴上绘制 6 个刻度标记符。这样可以简化后续运算。

现在让我们查看一下它的实现文件:

```
Plotter::Plotter(QWidget *parent)
    : QWidget(parent)
{
    setBackgroundRole(QPalette::Dark);
    setAutoFillBackground(true);
    setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    setFocusPolicy(Qt::StrongFocus);
    rubberBandIsShown = false;
```

```
    zoomInButton = new QToolButton(this);
    zoomInButton->setIcon(QIcon(":/images/zoomin.png"));
    zoomInButton->adjustSize();
    connect(zoomInButton, SIGNAL(clicked()), this, SLOT(zoomIn()));

    zoomOutButton = new QToolButton(this);
    zoomOutButton->setIcon(QIcon(":/images/zoomout.png"));
    zoomOutButton->adjustSize();
    connect(zoomOutButton, SIGNAL(clicked()), this, SLOT(zoomOut()));

    setPlotSettings(PlotSettings());
}
```

setBackgroundRole()调用告诉 QWidget 使用调色板中的“暗”分量作为重绘窗口部件的颜色，而不是使用“背景色”分量。这样就可以为 Qt 设置一种默认颜色，当把这个窗口部件重新改变成一个更大的尺寸时，甚至有可能是在 paintEvent()事件绘制那些新近显示的任意像素之前，就可以使用这种默认颜色来填充这些新的像素。我们也需要调用 setAutoFillBackground(true)来启用这一机制。（默认情况下，子窗口部件会从它们的父窗口部件那里继承相应的背景色。）

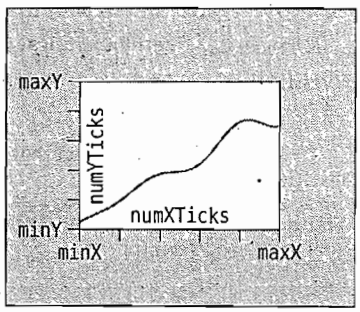


图 5.8　PlotSettings 的成员变量

setSizePolicy()调用可以把这个窗口部件的大小策略设置为在两个方向上都是 QSizePolicy::Expanding。这样就会告诉负责这个窗口布局的任意布局管理器，这个窗口部件可以放大，也可以缩小。这对于那些要占用很多屏幕空间的窗口部件来说，通常是一种比较典型的设置方法。在两个方向上，默认设置都是 QSizePolicy::Preferred，它的意思是指这个窗口部件的合适大小就是大小提示所给出的大小，但是如果有必要，也可以无限放大它，或者也可以一直把它缩小到大小提示给出的最小大小。

setFoucusPolicy(Qt::StrongFocus)调用可以让窗口部件通过单击或者通过按下 Tab 键而输入焦点。当 Plotter 获得焦点时，它将会接收由按键而产生的事件。Plotter 窗口部件可以处理一些按键：“+”用来放大图形，“-”用来缩小图形，以及还可以使用四个方向键来上、下、左、右地滚动图形。

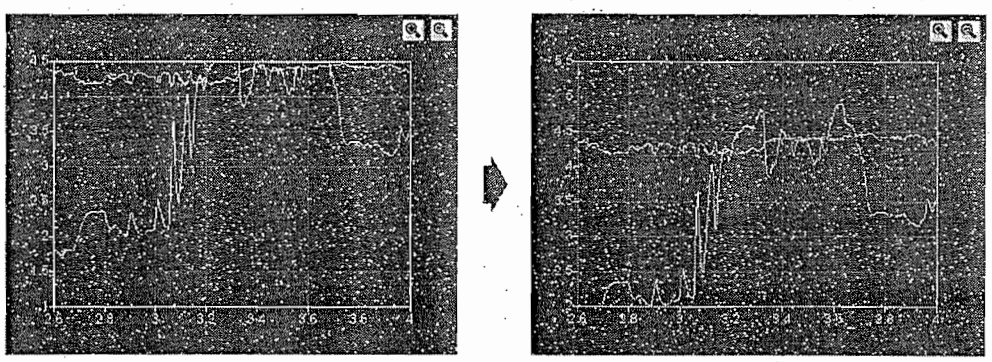


图 5.9　滚动 Plotter 窗口部件

同样还是在构造函数中，创建了两个 QToolButton，二者都有一个图标。允许用户通过这些按钮进行缩小和放大操作。这些按钮的图标存储在一个资源文件中，因此任何使用 Plotter 窗口部件的应用程序都需要在它们的 .pro 文件中加入这一条代码：

```
RESOURCES = plotter.qrc
```

资源文件与我们在 Spreadsheet 应用程序中曾经使用过的形式类似：

```
<RCC>
<qresource>
    <file>images/zoomin.png</file>
    <file>images/zoomout.png</file>
</qresource>
</RCC>
```

在这些按钮上调用的 adjustSize(),可以把它们的大小设置成大小提示所给定的大小。这些按钮没有放在布局中,相反,我们将自己动手把它们放在 Plotter 的重定义大小事件中。由于没有使用任何布局,所以必须通过对 QPushButton 的构造函数传递一个 this 指针来明确指定这些按钮所在的父对象。

最后,通过对 setPlotSettings()的调用就完成了初始化工作:

```
void Plotter::setPlotSettings(const PlotSettings &settings)
{
    zoomStack.clear();
    zoomStack.append(settings);
    curZoom = 0;
    zoomInButton->hide();
    zoomOutButton->hide();
    refreshPixmap();
}
```

setPlotSettings()函数用于指定显示绘图区时所用到的 PlotSettings。它可以被 Plotter 的构造函数调用,也可以被使用这个类的用户调用。绘图区开始时使用它的默认缩放级。用户每放大一次,都会创建一个新的 PlotSettings 实例,并且会将其放进缩放堆栈中。这个缩放堆栈由两个成员变量来表示:

- 类型为 QVector <PlotSettings> 的 zoomStack 保存不同的缩放级设置值。
- curZoom 在这个 zoomStack 中保存 PlotSettings 的当前索引值。

对 setPlotSettings()进行调用之后,该缩放堆栈会只包含一项,并且 Zoom In 和 Zoom Out 按钮也是隐藏的。只有当我们在 zoomIn()和 zoomOut()槽中对它们调用 show()的时候,才会把这些按钮显示出来。(一般情况下,在顶层窗口部件中调用 show()足以显示所有的子窗口部件。但是,当明确地对子窗口部件调用 hide()时,除非再次对它调用 show(),否则它就一直会处于隐藏状态。)

为了更新显示,refreshPixmap()调用是很有必要的。通常情况下,本可以调用 update(),但是这里的做法将会稍微有些不同,因为我们想让 QPixmap 在任意时刻都处于最新状态。在重新生成像素映射之后,refreshPixmap()会调用 update(),会把像素映射复制到窗口部件中。

```
void Plotter::zoomOut()
{
    if (curZoom > 0) {
        --curZoom;
        zoomOutButton->setEnabled(curZoom > 0);
        zoomInButton->setEnabled(true);
        zoomInButton->show();
        refreshPixmap();
    }
}
```

图形放大后,可以使用这个 zoomOut()槽进行缩小。它可以减小当前的缩放级数,并且可以根据这个图形是否还允许缩得更小来设置 Zoom Out 按钮是否生效。Zoom In 按钮会设置为生效并且显示出来,而通过调用 refreshPixmap()可以更新当前的显示结果。

```
void Plotter::zoomIn()
{
    if (curZoom < zoomStack.count() - 1) {
        ++curZoom;
```

```
        zoomInButton->setEnabled(curZoom < zoomStack.count() - 1);
        zoomOutButton->setEnabled(true);
        zoomOutButton->show();
        refreshPixmap();
    }
}
```

如果用户在此之前已经放大过图形并且又缩小了图形，那么用于下一缩放级数的 PlotSettings 将会放在这个缩放堆栈中，因而就可以放大图形。（此外，还有可能通过橡皮筋选择框来放大图形。）

这个槽可以把 curZoom 移动到缩放堆栈中更深的位置上，它会根据是否允许把这个图形放得更大来决定 Zoom In 按钮是生效还是无效，并且会启用和显示 Zoom Out 按钮。此外，会再次调用 refreshPixmap()，以便让这个绘图区能够使用到最新的缩放设置值。

```
void Plotter::setCurveData(int id, const QVector<QPointF> &data)
{
    curveMap[id] = data;
    refreshPixmap();
}
```

setCurveData()函数设置了用于给定曲线 ID 中的数据。如果一条曲线的 ID 与 curveMap 中已经存在的 ID 相同，那么将会用新的曲线数据替换原有的那些数据；否则，只是简单地插入新的曲线。curveMap 成员变量的类型为 QMap <int，QVector <QPointF>> 。

```
void Plotter::clearCurve(int id)
{
    curveMap.remove(id);
    refreshPixmap();
}
```

clearCurve()函数可以从 curveMap 中移除一条给定的曲线。

```
QSize Plotter::minimumSizeHint() const
{
    return QSize(6 * Margin, 4 * Margin);
}
```

minimumSizeHint()函数与 sizeHint()函数相似。就像 sizeHint()函数可以指定一个窗口部件的理想大小一样，minimumSizeHint()可以指定一个窗口部件理想的最小大小。布局决不会把一个窗口部件的大小修改为比它的最小大小提示还要小的大小。

我们返回的这个值是 300 × 200（因为 Margin = 50），这可以在 4 条边上留出一些空白区域，也可以为图形本身留出一些空间。如果小于这个大小，那么该绘图区就会显得太小了，也就没有什么用处了。

```
QSize Plotter::sizeHint() const
{
    return QSize(12 * Margin, 8 * Margin);
}
```

在 sizeHint()中，我们返回一个和边白常量 Margin 成比例的“理想”大小，它们会形成合适的3:2比例，就像 minimumSizeHint()中用到的比例一样。

至此，就完成了对 Plotter 公有函数和公有槽的回顾。现在，来看一下处于 protected 段中的事件处理器。

```
void Plotter::paintEvent(QPaintEvent * /* event */)
{
    QStylePainter painter(this);
    painter.drawPixmap(0, 0, pixmap);

    if (rubberBandIsShown) {
        painter.setPen(palette().light().color());
        painter.drawRect(rubberBandRect.normalized()
```

```
                                        .adjusted(0, 0, -1, -1));
    }

    if (hasFocus()) {
        QStyleOptionFocusRect option;
        option.initFrom(this);
        option.backgroundColor = palette().dark().color();
        painter.drawPrimitive(QStyle::PE_FrameFocusRect, option);
    }
}
```

通常情况下,paintEvent()是执行所有绘制任务的地方。但这里,图形区的所有绘制任务都在之前的 refreshPixmap()中完成了,所以只需简单地通过把该像素映射复制到窗口部件的(0,0)位置处来完成整个图形的绘制工作。

如果橡皮筋选择框可见,则可以把它绘制在图形区的顶部。我们从窗口部件的当前颜色组中选择"亮"分量作为画笔的颜色,这样可以确保绘制出的橡皮筋选择框能够与"暗"背景形成很好的反差。需要注意的是,我们是直接在窗口部件上绘制的,而并未改变脱屏像素映射。使用 QRect::normalized()可以确保这个矩形橡皮筋选择框的宽度和高度(如有必要,可以对换坐标)都是正值,并且 adjusted()可以把矩形的大小减去 1 个像素,以允许它具有 1 像素宽的轮廓。

如果 Plotter 拥有焦点,就会使用窗口部件风格的 drawPrimitive()函数绘制这个焦点选择框(focus rectangle),并使用 QStyle::PE_FrameFocusRect作为函数的第一个参数,使用 QStyleOptionFocusRect 对象作为函数的第二个参数。焦点选择框各个绘制选项的初始化会基于 Plotter 窗口部件[通过调用 initFrom()]。背景颜色需要明确给定。

当希望使用当前风格进行绘制时,则可以直接调用 QStyle 函数,例如:

```
style()->drawPrimitive(QStyle::PE_FrameFocusRect, &option, &painter,
                       this);
```

也可以使用 QStylePainter 代替常用的 QPainter,就像 Plotter 中所做的那样,并且使用 QStylePainter 会使绘制工作显得更加方便些。

QWidget::style()函数返回绘制窗口部件时应当使用的风格。在 Qt 中,窗口部件风格是 QStyle 的一个子类。内置的风格类型包括 QWindowsStyle、QWindowsXPStyle、QWindowsVistaStyle、QMotifStyle、QCDEStyle、QMacStyle、QPlastiqueStyle 以及 QCleanlooksStyle。每一种风格都重新实现了 QStyle 中的一些虚函数,它们会以该风格所模拟平台的正确方式来执行绘制操作。QStylePainter 的 drawPrimitive()函数调用与 QStyle 中函数同名的函数,这样就可以用来绘制像面板、按钮和焦点选择框这样的一些"基本元素"(primitive element)。在一个应用程序中,所有窗口部件的风格(QApplication::style())通常都是相同的,但是也可以在每个窗口部件的基础上使用 QWidget::setStyle()对它们的风格进行重新定义。

通过子类化 QStyle,就有可能定义一种新的自定义风格。这样就可以为一个或者一组应用程序设置一种与众不同的外观,就像第 19 章中将看到的那样。通常情况下,尽管我们认为使用目标平台所提供的本地化外观是一种明智的选择,但如果你乐于探索的话,Qt 还是为你提供了很大的自由度。

Qt 的内置窗口部件几乎都只依赖于 QStyle 来绘制自身。这就是为什么在 Qt 支持的所有平台上,它们看起来都像原有平台上的窗口部件一样。要获得与风格相关的自定义窗口部件,既可以通过使用 QStyle 绘制它们自身的方式,也可以通过把 Qt 的内置窗口部件作为子窗口部件的方式。对于 Plotter,则使用这两种方式的组合:使用 QStyle(借助 QStylePainter)绘制焦点选择框,而 Zoom In 和 Zoom Out 按钮则使用 Qt 的内置窗口部件来完成。

```
void Plotter::resizeEvent(QResizeEvent * /* event */)
{
    int x = width() - (zoomInButton->width()
                       + zoomOutButton->width() + 10);
    zoomInButton->move(x, 5);
    zoomOutButton->move(x + zoomInButton->width() + 5, 5);
    refreshPixmap();
}
```

只要一改变 Plotter 窗口部件的大小，Qt 就会产生一个“重定义大小”事件。这里重新实现了 resizeEvent()，它把 Zoom In 和 Zoom Out 按钮放置在 Plotter 窗口部件的右上角。

我们把 Zoom In 和 Zoom Out 按钮一个挨一个地放好，它们之间有 5 个像素的间隔，并且距离它们的父窗口部件的上边缘和右边缘各有 5 个像素。

如果想把这些按钮总是放在图形的左上角，其坐标是(0,0)，那么只需在 Plotter 的构造函数中简单修改一下就可以了。但是，如果想追踪的是图形的右上角，那么此处的坐标就取决于窗口部件的大小。正是因为这一点，重新实现 resizeEvent()并且把这些按钮的位置放在那里就显得很有必要了。

在 Plotter 的构造函数中，没有为按钮设置任何位置。但这不并是什么问题，因为在一个窗口部件第一次显示之前，Qt 总是会自动产生一个重定义大小事件。

除了通过重新实现 resizeEvent()并且手动摆放这些子窗口部件这一方法之外，还有可能使用的另外一种方法是使用布局管理器(例如，QGridLayout)。使用布局可能会稍微有些麻烦，并且可能会消耗更多的资源。另一方面，能够按照从右到左的方式来处理布局可能会显得更漂亮些，而且这对于像阿拉伯语和希伯来语的那些语言来说也是很有必要的。

最后，调用 refreshPixmap()，以按照新的尺寸大小重新绘制像素映射。

```
void Plotter::mousePressEvent(QMouseEvent *event)
{
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);

    if (event->button() == Qt::LeftButton) {
        if (rect.contains(event->pos())) {
            rubberBandIsShown = true;
            rubberBandRect.setTopLeft(event->pos());
            rubberBandRect.setBottomRight(event->pos());
            updateRubberBandRegion();
            setCursor(Qt::CrossCursor);
        }
    }
}
```

当用户按下鼠标左键，就开始显示一个橡皮筋选择框。这样会产生一系列事件，像把 rubberBandIsShown 设置为 true，以当前鼠标指针所在的位置初始化 rubberBandRect，调用一个绘制该橡皮筋选择框的绘制事件以及把鼠标光标修改成十字形光标等。

rubberBandRect 变量的类型是 QRect。QRect 既可以通过 4 个分量(x, y, *width*, *height*)定义——这里的(x, y)是它的左上角的点的位置坐标，*width* × *height* 是这个矩形的大小，也可以使用一个左上角和一个右下角两个点而形成的一个坐标对来定义。这里使用的表示方法就是坐标对表示法。我们把用户点击的点既当作橡皮筋选择框的左上角坐标又把它当作右下角坐标。然后，调用 updateRubberBandRegion()来对橡皮筋选择框所覆盖的(最小)区域进行强制重绘。

Qt 提供了两种用于控制鼠标光标形状的机制：

- 当鼠标悬停在某个特殊的窗口部件上时，QWidget::setCursor()可以设置它所使用的光标形

状。如果没有为窗口部件专门设置光标,那么就会使用它的父窗口部件中的光标。顶层窗口部件的默认光标是箭头光标。

- 对于整个应用程序中所使用的光标形状,可以通过 QApplication::setOverrideCursor()进行设置,它会把不同窗口部件中的光标形状全部覆盖掉,直到调用 restoreOverrideCursor()。

在第 4 章中,就调用了以 Qt::WaitCurosr 为参数的 QApplication::setOverrideCursor(),它把应用程序的光标变成了标准的等待光标(wait cursor)。

```
void Plotter::mouseMoveEvent(QMouseEvent *event)
{
    if (rubberBandIsShown) {
        updateRubberBandRegion();
        rubberBandRect.setBottomRight(event->pos());
        updateRubberBandRegion();
    }
}
```

当用户按下鼠标左键并移动鼠标光标的时候,会首先调用 updateRubberBandRegion()来预约一个绘制事件,由其对橡皮筋选择框所在的区域进行重新绘制。然后,重新计算 rubberBandRect 来说明这次鼠标移动的距离。最后,调用 updateRubberBandRegion()再一次对绘制橡皮筋选择框已经移动到的区域进行重新计算。这样做,可以有效地擦除橡皮筋选择框,并且可以在新的坐标系中重新绘制它。

如果用户向上或者向左移动鼠标,那么 rubberBandRect 名义上的右下角看起来就好像跑到了左上角的上面或者左面了。如果发生这种情况,那么这个 QRect 就会具有一个负的宽度或者高度值。因此在 paintEvent()中使用 QRect::normalized(),从而可以对它的左上角和右下角坐标进行调整,以确保能够获得非负的宽度值或者高度值。

```
void Plotter::mouseReleaseEvent(QMouseEvent *event)
{
    if ((event->button() == Qt::LeftButton) && rubberBandIsShown) {
        rubberBandIsShown = false;
        updateRubberBandRegion();
        unsetCursor();

        QRect rect = rubberBandRect.normalized();
        if (rect.width() < 4 || rect.height() < 4)
            return;
        rect.translate(-Margin, -Margin);

        PlotSettings prevSettings = zoomStack[curZoom];
        PlotSettings settings;
        double dx = prevSettings.spanX() / (width() - 2 * Margin);
        double dy = prevSettings.spanY() / (height() - 2 * Margin);
        settings.minX = prevSettings.minX + dx * rect.left();
        settings.maxX = prevSettings.minX + dx * rect.right();
        settings.minY = prevSettings.maxY - dy * rect.bottom();
        settings.maxY = prevSettings.maxY - dy * rect.top();
        settings.adjust();

        zoomStack.resize(curZoom + 1);
        zoomStack.append(settings);
        zoomIn();
    }
}
```

当用户释放鼠标左键时,会擦除这个橡皮筋选择框,并且把光标恢复成标准的箭头光标。如果橡皮筋选择框不小于 4×4,那么就执行缩放操作。如果橡皮筋选择框小于 4×4,看来应当是用户错误地点击了窗口部件,或者仅仅是为了让窗口部件获得焦点,所以就什么也不做。

这段执行缩放的代码稍微有一点复杂。这是因为需要同时处理窗口部件坐标系和绘图区坐

标系。这里所做的绝大多数工作都是为了把 rubberBandRect 的坐标从窗口部件坐标系转换成绘图区坐标系。这一转换过程一旦完成，就可以调用 PlotSettings::adjust()圆整这些数据，并且为每根坐标轴找出一个合适的刻度标记符个数。图 5.10 和图 5.11 描述了这一处理过程。

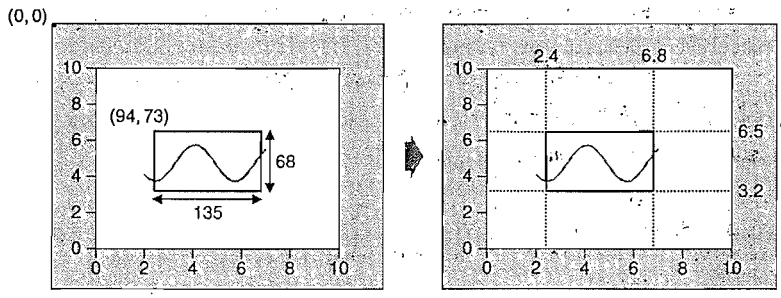


图 5.10　把橡皮筋选择框的坐标从窗口部件坐标系转换成绘图区坐标系

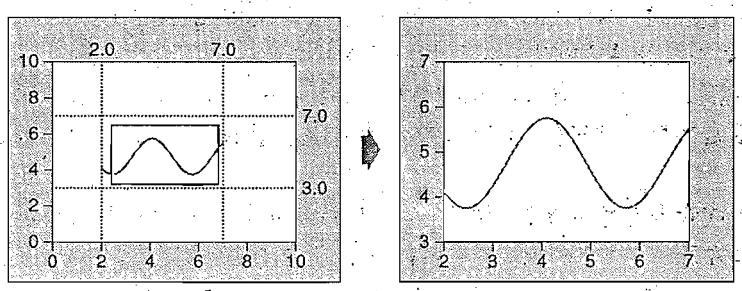


图 5.11　调整绘图区坐标系并且放大橡皮筋选择框

然后，执行缩放操作。这一缩放工作的实现是通过把刚刚计算过的 PlotSettings 放到缩放堆栈的顶端并且调用 zoomIn()来完成的。

```
void Plotter::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
    case Qt::Key_Plus:
        zoomIn();
        break;
    case Qt::Key_Minus:
        zoomOut();
        break;
    case Qt::Key_Left:
        zoomStack[curZoom].scroll(-1, 0);
        refreshPixmap();
        break;
    case Qt::Key_Right:
        zoomStack[curZoom].scroll(+1, 0);
        refreshPixmap();
        break;
    case Qt::Key_Down:
        zoomStack[curZoom].scroll(0, -1);
        refreshPixmap();
        break;
    case Qt::Key_Up:
        zoomStack[curZoom].scroll(0, +1);
        refreshPixmap();
        break;
    default:
        QWidget::keyPressEvent(event);
    }
}
```

当用户按下一个键并且 Plotter 窗口部件拥有焦点时,就会调用 keyPressEvent()函数。在这里,我们重新实现该函数以响应 6 个按键:+、-以及向上、向下、向左和向右 4 个方向键。如果用户按下的是一个无法处理的键,那么就调用基类的实现。为简便起见,我们忽略了 Shift、Ctrl 和 Alt 这些修饰键,可以通过 QKeyEvent::modifiers()来获得它们的状态。

```
void Plotter::wheelEvent(QWheelEvent *event)
{
    int numDegrees = event->delta() / 8;
    int numTicks = numDegrees / 15;

    if (event->orientation() == Qt::Horizontal) {
        zoomStack[curZoom].scroll(numTicks, 0);
    } else {
        zoomStack[curZoom].scroll(0, numTicks);
    }
    refreshPixmap();
}
```

当转动鼠标滚轮(wheel)时,就会产生滚轮事件。绝大多数鼠标只提供一个垂直滚轮,但是也有一些鼠标还提供了另外一个水平滚轮。对这两种滚轮 Qt 都可以支持。滚轮事件会到达那些拥有焦点的窗口部件。delta()函数可以返回一个距离,它等于滚轮旋转角度的 8 倍。鼠标通常都以 15 度作为步长。这里,通过修改缩放堆栈上最顶端的元素来改变所需的标记符的数量,并且使用 refreshPixmap()更新显示。

鼠标滚轮最为常见的用法是用它来滚动一个滚动条。当使用 QScrollArea(将会在第 6 章中讲到)来提供一些滚动条的时候,QScrollArea 会自动处理鼠标的滚动事件,所以不需要重新实现 wheelEvent()。

这样就完成了对事件处理器的实现。现在来看一下私有函数。

```
void Plotter::updateRubberBandRegion()
{
    QRect rect = rubberBandRect.normalized();
    update(rect.left(), rect.top(), rect.width(), 1);
    update(rect.left(), rect.top(), 1, rect.height());
    update(rect.left(), rect.bottom(), rect.width(), 1);
    update(rect.right(), rect.top(), 1, rect.height());
}
```

updateRubberBand()函数会在 mousePressEvent()、mouseMoveEvnet()和 mouseReleaseEvent()中得到调用,用来擦除或者重新绘制橡皮筋选择框。这个函数由 4 个 update()调用组成,它们为橡皮筋选择框(由两条垂直线和两条水平线构成)所覆盖的 4 个小矩形区域调用一个绘制事件。

```
void Plotter::refreshPixmap()
{
    pixmap = QPixmap(size());
    pixmap.fill(this, 0, 0);

    QPainter painter(&pixmap);
    painter.initFrom(this);
    drawGrid(&painter);
    drawCurves(&painter);
    update();
}
```

refreshPixmap()函数把绘图区重新绘制到脱屏像素映射上,并且对显示加以更新。我们把像素映射的大小调整为与窗口部件的大小一样,并且使用窗口部件的擦除颜色来填充该像素映射。这个颜色就是调色板中的"暗"分量,因为是在 Plotter 的构造函数中调用 setBackgroundRole()的。如果背景色是一个非实心画笔,QPixmap::fill()就需要知道窗口部件中的偏移量,像素映射需要在那里恰好与画笔图案相对齐。这里,像素映射与整个窗口部件相对应,所以就指定位置(0,0)。

然后，创建一个 QPainter 在这个像素映射上进行绘制。initFrom()调用可以设置 painter 所使用的画笔、背景色和字体，这些都与 Plotter 窗口部件中的一样。接下来，调用 drawGrid()和 drawCurves()来执行绘制。最后，调用 update()，从而为整个窗口部件预约一个绘制事件。在 97 页的 paintEvent()函数中，已经将把这个像素映射复制到窗口部件中。

```
void Plotter::drawGrid(QPainter *painter)
{
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
    if (!rect.isValid())
        return;

    PlotSettings settings = zoomStack[curZoom];
    QPen quiteDark = palette().dark().color().light();
    QPen light = palette().light().color();

    for (int i = 0; i <= settings.numXTicks; ++i) {
        int x = rect.left() + (i * (rect.width() - 1)
                               / settings.numXTicks);
        double label = settings.minX + (i * settings.spanX()
                                        / settings.numXTicks);
        painter->setPen(quiteDark);
        painter->drawLine(x, rect.top(), x, rect.bottom());
        painter->setPen(light);
        painter->drawLine(x, rect.bottom(), x, rect.bottom() + 5);
        painter->drawText(x - 50, rect.bottom() + 5, 100, 20,
                          Qt::AlignHCenter | Qt::AlignTop,
                          QString::number(label));
    }
    for (int j = 0; j <= settings.numYTicks; ++j) {
        int y = rect.bottom() - (j * (rect.height() - 1)
                                 / settings.numYTicks);
        double label = settings.minY + (j * settings.spanY()
                                        / settings.numYTicks);
        painter->setPen(quiteDark);
        painter->drawLine(rect.left(), y, rect.right(), y);
        painter->setPen(light);
        painter->drawLine(rect.left() - 5, y, rect.left(), y);
        painter->drawText(rect.left() - Margin, y - 10, Margin - 5, 20,
                          Qt::AlignRight | Qt::AlignVCenter,
                          QString::number(label));
    }
    painter->drawRect(rect.adjusted(0, 0, -1, -1));
}
```

drawGrid()函数绘制曲线和坐标轴后面的网格。绘制网格的区域是通过 rect 给定的。如果窗口部件不够大，不能容纳下这个图形，就立即返回。

第 1 个 for 循环绘制了网格的垂直线和沿 x 轴方向上的标记符。第 2 个 for 循环绘制了网格的水平线和沿 y 轴方向上的标记符。最后，沿空白区域绘制一个长方形。drawText()函数用于绘制两个坐标轴上与记号相对应的那些数字。

对于 drawText()的调用遵循下面的语法形式：

```
painter->drawText(x, y, width, height, alignment, text);
```

其中，(*x*, *y*, *width*, *height*)定义了一个矩形，*alignment* 是在这个矩形中文本的位置，而 *text* 就是要绘制的文本。在本例中，已经人工计算出了用于绘制文本的矩形。另外一种具有更好适应性的方法或许应当是使用 QFontMetrics，其中包含了文本边界框的计算。

```
void Plotter::drawCurves(QPainter *painter)
{
    static const QColor colorForIds[6] = {
        Qt::red, Qt::green, Qt::blue, Qt::cyan, Qt::magenta, Qt::yellow
    };
```

```
    PlotSettings settings = zoomStack[curZoom];
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
    if (!rect.isValid())
        return;

    painter->setClipRect(rect.adjusted(+1, +1, -1, -1));

    QMapIterator<int, QVector<QPointF> > i(curveMap);
    while (i.hasNext()) {
        i.next();

        int id = i.key();
        QVector<QPointF> data = i.value();
        QPolygonF polyline(data.count());
        for (int j = 0; j < data.count(); ++j) {
            double dx = data[j].x() - settings.minX;
            double dy = data[j].y() - settings.minY;
            double x = rect.left() + (dx * (rect.width() - 1)
                                      / settings.spanX());
            double y = rect.bottom() - (dy * (rect.height() - 1)
                                        / settings.spanY());
            polyline[j] = QPointF(x, y);
        }
        painter->setPen(colorForIds[uint(id) % 6]);
        painter->drawPolyline(polyline);
    }
}
```

drawCurves()函数在网格上绘制这些曲线。我们从调用 setClipRect()开始,它为包含这些曲线(边白和包围绘图区的框架除外)的矩形设置 QPainter 剪辑区。然后,QPainter 将会忽略在这个区域之外的像素绘制操作。

接下来,使用 Java 形式的迭代器遍历所有曲线,并且对于每一条曲线,都要遍历构成它的每一个 QPointF 值。我们调用迭代器的 key()函数获取曲线的 ID,并且用它的 value()函数获取相应曲线的 QVector<QPointF>类型数据。内部的 for 循环把每个 QPointF 都从绘图区坐标系转换到窗口部件坐标系,并且把它们保存到 polyline 变量中。

一旦已经将一条曲线全部点的坐标都转换成了窗口部件中的坐标,就可以为用于这条曲线的画笔设置颜色(使用预定义的颜色集之一),并且调用 drawPolyline()绘制一条经过该曲线上所有点的直线。

这就是完整的 Plotter 类。现在只剩下 PlotSettings 中的一些函数了。

```
PlotSettings::PlotSettings()
{
    minX = 0.0;
    maxX = 10.0;
    numXTicks = 5;

    minY = 0.0;
    maxY = 10.0;
    numYTicks = 5;
}
```

PlotSettings 的构造函数对两个坐标轴进行了初始化,把它们的范围设置成从 0 到 10,其间使用了 5 个标记符。

```
void PlotSettings::scroll(int dx, int dy)
{
    double stepX = spanX() / numXTicks;
    minX += dx * stepX;
    maxX += dx * stepX;

    double stepY = spanY() / numYTicks;
    minY += dy * stepY;
    maxY += dy * stepY;
}
```

scroll()函数使用两个记号之间的距离乘以一个给定数字的方式来增加(或者减少)minX、maxX、minY 和 maxY 的值。这个函数在 Plotter::keyPressEvent()中用来实现滚动功能。

```
void PlotSettings::adjust()
{
    adjustAxis(minX, maxX, numXTicks);
    adjustAxis(minY, maxY, numYTicks);
}
```

从 mouseReleaseEvent()中调用的 adjust()函数,用来把 minX、maxX、minY 和 maxY 圆整成"合适的"数值,并且用于决定每一个坐标轴上应当使用的标记符个数。私有函数 adjustAxis()每次可以处理一个坐标轴。

```
void PlotSettings::adjustAxis(double &min, double &max, int &numTicks)
{
    const int MinTicks = 4;
    double grossStep = (max - min) / MinTicks;
    double step = std::pow(10.0, std::floor(std::log10(grossStep)));

    if (5 * step < grossStep) {
        step *= 5;
    } else if (2 * step < grossStep) {
        step *= 2;
    }

    numTicks = int(std::ceil(max / step) - std::floor(min / step));
    if (numTicks < MinTicks)
        numTicks = MinTicks;
    min = std::floor(min / step) * step;
    max = std::ceil(max / step) * step;
}
```

adjustAxis()函数把它的 min 和 max 参数转换成"合适的"数值,并且在给定的[min,max]范围内计算出合适的标记符个数,然后把它的 numTicks 参数设置成该值。因为 adjustAxis()需要对变量(minX、maxX、numXTicks,等等)进行实际修改而不仅仅只是复制,所以它的参数都是非常量型引用。

adjustAxis()中的绝大多数代码都只是简单地试图确定两个记号之间适当距离的合适数值(这里就是"步长")。为了沿每一根轴都可以获得这样的合适数值,必须小心地选择这个步长。例如,一个步长为 3.8 的数值会导致这根轴的长度应当是 3.8 的若干倍,但是这样的数值很难使人认同它。对于以十进制为标记的轴,"合适的"步长值应该具有像 10^n、2×10^n 或者 5×10^n 这样的数值形式。

一开始,我们先计算"总步长",它是该步长最大值的一种形式。然后,找到一个小于或等于这个总步长并且形式为 10^n 的对应值。对它进行以 10 为基数的对数运算,然后对它取整,再以 10 为底、以前面取整所得的整数为指数,从而形成幂的形式。例如,如果总步长为 236,则计算 log 236 = 2.37291…。然后,对它取整得到整数 2,这样就可以得到 $10^2 = 100$,也就是具有形式为 10^n 的一个候选步长值。

一旦获得了第一个步长候选值,就可以使用它计算另外两种形式的候选值:2×10^n 和 5×10^n。根据前面的计算示例,这另外的两个候选值分别为 200 和 500。而 500 比总步长要大,所以不能使用它。200 则比 236 要小,所以在这个例子中,我们使用 200 作为步长值。

利用这个步长值,就可以很容易地推算出 numTicks、min 和 max 的值。新的 min 值是小于初始 min 值的最近的步长倍数值,而新的 max 值大于初始 max 的最近的步长倍数值。新的 numTicks 是取整后的 min 和 max 之间的距离个数。例如,如果在这个函数中输入的 min 为 240 并且 max 为 1184,那么新的范围就变为[200,1200],有 5 个标记符。

这种算法在有些情况下给出的并不是最佳结果。发表在“Graphics Gems”(Morgan Kaufmann, 1990)上 Paul S. Heckbert 的文章“Nice Numbers for Graph Labels”中,描述了一个复杂的算法。

通过这一章,就结束了本书的第一部分。这一章解释了如何对已经存在的 Qt 窗口部件进行自定义,并且说明了如何使用 QWidget 作为基类来构建窗口部件的方法。第 2 章讲解了如何使用已有的窗口部件来组建新的窗口部件,而在第 6 章还会对这一主题做进一步探索。

到此为止,我们利用 Qt 已经足以编写出完整的图形用户界面应用程序。在第二部分和第三部分,将会更进一步探索 Qt,以使我们能够更为充分地利用 Qt 的强大功能。

第二部分　Qt 中级

第 6 章　布局管理
第 7 章　事件处理
第 8 章　二维图形
第 9 章　拖放
第 10 章　项视图类
第 11 章　容器类
第 12 章　输入与输出
第 13 章　数据库
第 14 章　多线程
第 15 章　网络
第 16 章　XML
第 17 章　提供在线帮助

第6章 布局管理

放置在窗体中的每一个窗口部件都必须给定一个合适的大小和位置。Qt 提供了多个用于在窗体中摆放窗口部件的类：QHBoxLayout、QVBoxLayout、QGridLayout 和 QStackLayout。这些类简单易用，几乎每个 Qt 开发人员都会用到它们——或者直接在源代码中，或者通过 Qt 设计师。

使用 Qt 布局类的另外一个原因是：它们可以确保窗体能够自动适应于不同的字体、语言和系统平台。如果用户改变了系统的字体设置，那么该应用程序的窗体将能够立刻做出响应，并在必要的情况下重新改变自己的大小。并且如果将应用程序的用户接口翻译成了另外一种语言，那么这些布局类就会考虑窗口部件中翻译的内容，以尽量避免文本被截断的现象发生。

可以执行布局管理功能的其他类还有 QSplitter、QScrollArea、QMainWindow 和 QMdiArea。这些类所拥有的共同点在于它们提供了一种用户可以灵活掌控的布局方式。例如，QSplitter 就提供了一个切分窗口拖动条(splitter bar)，通过拖拽它，用户可以改变窗口部件的大小。QMdiArea 则为多文档界面(multiple document interface，MDI)——可以在一个应用程序主窗口中同时显示多个文档的方法——提供了支持。因为它们经常适合用作布局类的替换方式，所以将在本章对它们进行详细介绍。

6.1 在窗体中摆放窗口部件

一共有三种基本方法用于管理窗体上子窗口部件的布局：绝对位置法、人工布局法和布局管理器法。我们将以图 6.1 中所给出的 Find File 对话框为例来依次说明这三种方法。

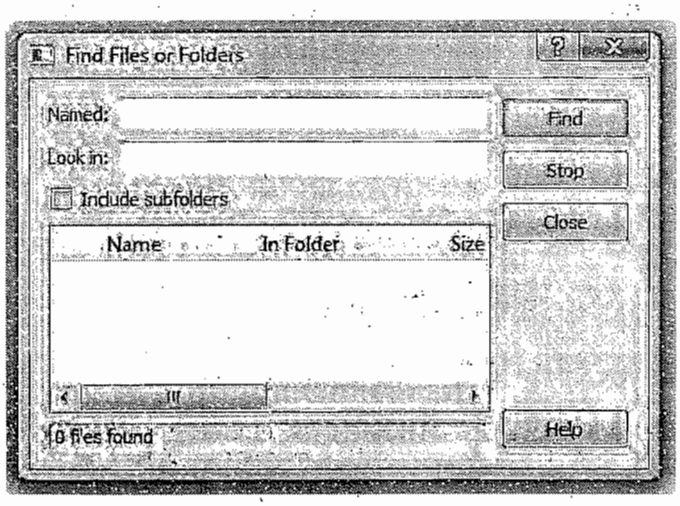


图 6.1 Find File 对话框

绝对位置法是一种最原始的摆放窗口部件的方法。这可以通过对窗体的各个子窗口部件分配固定的大小和位置以及对窗体分配固定的大小实现。这里给出的是使用了绝对位置法的 FindFileDialog 构造函数，它看起来具有如下形式：

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 200, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 200, 25);
    subfoldersCheckBox->setGeometry(9, 71, 256, 23);
    tableWidget->setGeometry(9, 100, 256, 100);
    messageLabel->setGeometry(9, 206, 256, 25);
    findButton->setGeometry(271, 9, 85, 32);
    stopButton->setGeometry(271, 47, 85, 32);
    closeButton->setGeometry(271, 84, 85, 32);
    helpButton->setGeometry(271, 199, 85, 32);

    setWindowTitle(tr("Find Files or Folders"));
    setFixedSize(365, 240);
}
```

绝对位置法有很多缺点：

- 用户无法改变窗口的大小。
- 如果用户选择了一种不常用的大字体，或者当应用程序被翻译成另外一种语言时，也许会把一些文本截断。
- 对于某些风格的平台，这些窗口部件可能会具有并不合适的尺寸大小。
- 必须人工计算这些位置和大小。这样做不仅非常枯燥乏味且极易出错，并且还会让后期的维护工作变得痛苦万分。

代替绝对位置法的另外一种方法是人工布局法。尽管还是需要给定窗口部件的绝对位置，但是利用人工布局方法给定的大小尺寸总是可以和窗口的大小成比例，这比完全依靠手写代码要好得多了。通过重新实现窗体的 resizeEvent()函数，该函数可以设置窗体中的子窗口部件的几何形状，就可以实现人工布局法。

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    setMinimumSize(265, 190);
    resize(365, 240);
}

void FindFileDialog::resizeEvent(QResizeEvent * /* event */)
{
    int extraWidth = width() - minimumWidth();
    int extraHeight = height() - minimumHeight();

    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 100 + extraWidth, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 100 + extraWidth, 25);
    subfoldersCheckBox->setGeometry(9, 71, 156 + extraWidth, 23);
    tableWidget->setGeometry(9, 100, 156 + extraWidth,
                             50 + extraHeight);
    messageLabel->setGeometry(9, 156 + extraHeight, 156 + extraWidth,
                              25);
    findButton->setGeometry(171 + extraWidth, 9, 85, 32);
    stopButton->setGeometry(171 + extraWidth, 47, 85, 32);
    closeButton->setGeometry(171 + extraWidth, 84, 85, 32);
    helpButton->setGeometry(171 + extraWidth, 149 + extraHeight, 85,
                            32);
}
```

在 FindFileDialog 构造函数中,我们把窗体的最小大小设置为 265 × 190,把它的初始大小设置为 365 × 240。在 resizeEvent()处理器中,将多余的任意空间都留给了那些希望变长或者变高的窗口部件。这样就确保了当用户重新改变该窗体大小时,可以稳步地缩放它。

就像前面的绝对位置法一样,人工布局法需要程序员对很多手写代码中的常量进行计算。像这样来编写代码的确是一件烦人的事情,尤其是在改变设计的时候更是如此,并且它仍旧存在文本会被截断的危险。通过考虑子窗口部件的大小提示,就可以避免这种风险,但是那样将会使代码变得更为复杂。

对于在窗体上如何摆放窗口部件,最简便易行的解决方法是使用 Qt 的布局管理器。布局管理器会为每种类型的窗口部件提供一些合理的默认值,并且也会考虑每一个窗口部件的大小提示,这些大小提示又通常会取决于该窗口部件的字体、风格和内容。布局管理器也会充分考虑其最小和最大尺寸,并且会自动通过调整布局来响应字体的变化、内容的改变以及窗口大小的调整。一种可以改变大小的 Find File 对话框新版本如图 6.2 所示。

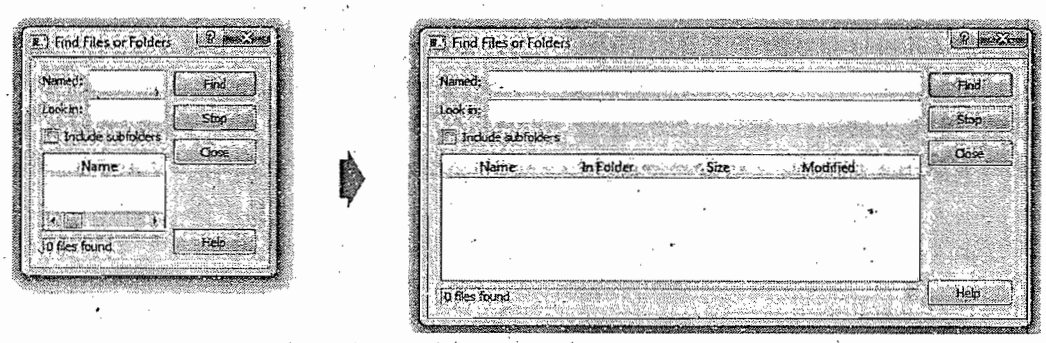


图 6.2 重定义可变尺寸对话框的大小

最为重要的三种布局管理器是:QHBoxLayout、QVBoxLayout 和 QGridLayout。这三个类从 QLayout 中派生出来,而 QLayout 类为布局提供了基本框架。这三个类可以得到 Qt 设计师的完全支持,并且也可以直接在代码中使用它们。

以下是使用布局管理器的 FindFileDialog 的代码:

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    QGridLayout *leftLayout = new QGridLayout;
    leftLayout->addWidget(namedLabel, 0, 0);
    leftLayout->addWidget(namedLineEdit, 0, 1);
    leftLayout->addWidget(lookInLabel, 1, 0);
    leftLayout->addWidget(lookInLineEdit, 1, 1);
    leftLayout->addWidget(subfoldersCheckBox, 2, 0, 1, 2);
    leftLayout->addWidget(tableWidget, 3, 0, 1, 2);
    leftLayout->addWidget(messageLabel, 4, 0, 1, 2);

    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(stopButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch();
    rightLayout->addWidget(helpButton);

    QHBoxLayout *mainLayout = new QHBoxLayout;
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);
```

```
    setLayout(mainLayout);
    setWindowTitle(tr("Find Files or Folders"));
}
```

通过一个 QHBoxLayout、一个 QGridLayout 和一个 QVBoxLayout，该布局就得到处理，如图 6.3 所示。通过外面的 QHBoxLayout，就把左侧的 QGridLayout 和右侧的 QVBoxLayout 一个挨一个地放在了一起。对话框周围的边白和两个子窗口部件之间的间隔均被设置为默认值，该值取决于当前窗口部件的风格，但通过 QLayout::setContentsMargins ()和 QLayout::setSpacing()，可以对它们的值进行修改。

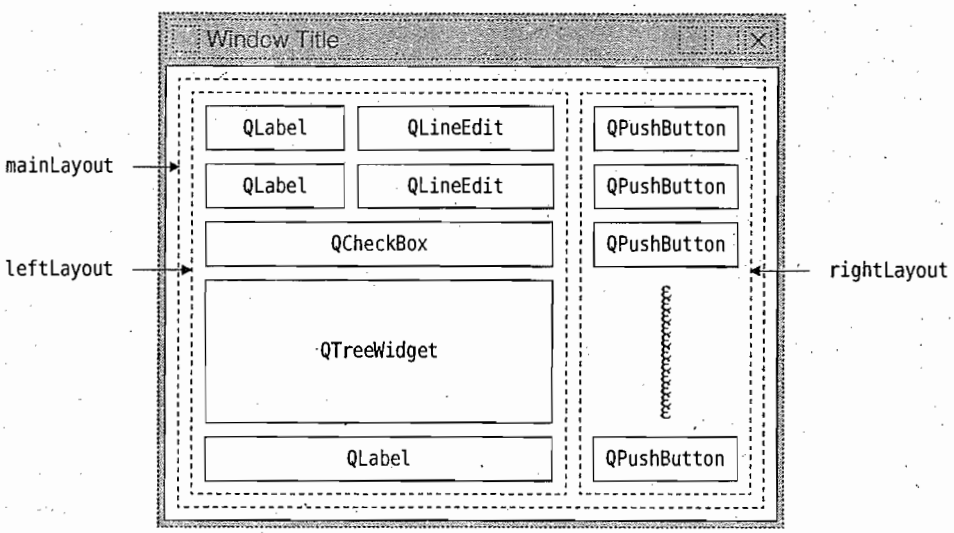


图 6.3 Find File 对话框中的布局

在 Qt 设计师中，也可以通过可视化的方式创建出同样的对话框，即：通过把这些子窗口部件摆放在大致的地方，选择那些需要摆放在一起的窗口部件，然后再单击 Form→Lay Out Horizontally，Form→Lay Out Vertically，或者 Layout→Lay Out in a Grid 即可。在第 2 章，我们就是使用这种方法创建了用于 Spreadsheet 应用程序的 Go to Cell 对话框和 Sort 对话框的。

QHBoxLayout 和 QVBoxLayout 的用法相当简单明了，但 QGridLayout 的用法则稍微有些麻烦。QGridLayout 的工作基于一个二维单元格。在这个布局中，左上角的 QLabel 的位置是(0,0)，而与之相应的 QLineEdit 的位置是(0,1)。QCheckBox 占用两列，也就是位置为(2,0)和(2,1)的两个单元格。在它下面的 QTreeWidget 和 QLable 也占用两列。对于 QGridLayout::addWidget()的调用遵循如下的语法形式：

```
layout->addWidget(widget, row, column, rowSpan, columnSpan);
```

其中，*widget* 是要插入到布局中的子窗口部件，(*row*, *column*)是由该窗口部件所占用的左上角单元格，*rowSpan* 是该窗口部件要占用的行数，而 *columnSpan* 是该窗口部件要占用的列数。如果省略了这些参数，则参数 *rowSpan* 和 *columnSpan* 将会取默认值 1。

addStretch()调用告诉垂直布局管理器，它会占满布局中这一处的空间。通过添加一个拉伸项，就相当于已经告诉布局管理器，需要占用 Close 按钮和 Help 按钮之间的全部多余空间。在 Qt 设计师中，可以通过插入一个分隔符(spacer)来达到同样的效果，分隔符会显示成蓝色的“弹簧”形状。

对于目前所探讨的问题，使用布局管理器的确为我们提供了很多额外的好处。如果往布局中

添加一个窗口部件或者从布局中移除一个窗口部件,布局都会自动适应所产生的这些新情况。如果对一个子窗口部件调用了 hide()或者 show(),也同样能够做到自动适应。如果一个子窗口部件的大小提示发生了变化,布局将会自动进行调整,从而把新的大小提示考虑进去。还有,布局管理器也会自动根据窗体中子窗口部件的最小大小提示和大小提示,从总体上为这个窗体设置一个最小尺寸。

在迄今为止所给出的每一个例子中,我们只是简单地把窗口部件放置到布局中,并且使用一定的分隔符元素(拉伸因子)来占用任何多余的空间。但在某些情况下,由此形成的布局看起来可能还不是我们最想要的形式。在这些情形中,可以通过改变要摆放的窗口部件的大小尺寸策略和大小提示来调整布局。

一个窗口部件的大小策略会告诉布局系统应该如何对它进行拉伸或者压缩。Qt 为它所有的内置窗口部件都提供了合理的默认大小策略值,但是由于不可能为每一种可能产生的布局都提供唯一的默认值,所以在一个窗体中,开发人员改变它上面的一个或两个窗口部件的大小策略是非常普遍的现象。一个 QSizePolicy 既包含一个水平分量也包含一个垂直分量。以下是一些最为常用的取值:

- Fixed 的意思是该窗口部件不能被拉伸或者压缩。窗口部件的大小尺寸总是保持为其大小提示的尺寸。
- Minimum 的意思是该窗口部件的大小提示就是它的最小大小。再不能把窗口部件压缩到比这个大小提示还要小的大小,但是如有必要,可以拉伸它来填充尽可能多的空间。
- Maximum 的意思是该窗口部件的大小提示就是它的最大大小。但是可以把该窗口部件压缩成它的最小大小提示的尺寸。
- Preferred 的意思是该窗口部件的大小提示就是它比较合适的大小。但是如果需要,还是可以对该窗口部件进行拉伸或者压缩。
- Expanding 的意思是可以拉伸或者压缩该窗口部件,并且它特别希望能够变长变高。

图 6.4 使用一个显示"Some Text"文本的 QLabel 为例,对这些不同的大小策略的含义进行了概括。

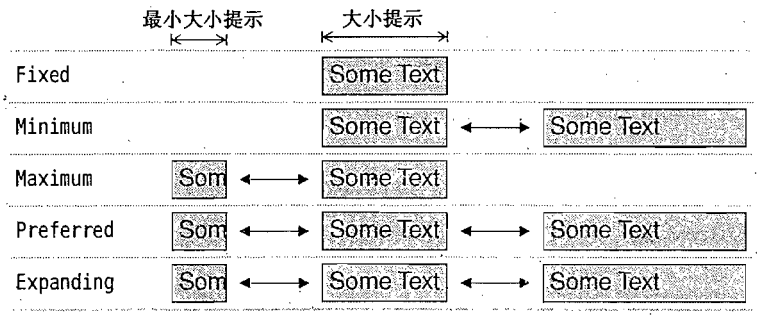


图 6.4 不同大小策略的含义

在图 6.4 中,Preferred 和 Expanding 描述成了同样的效果。但是,它们之间到底有何不同呢? 在重新改变一个既包含有 Preferred 又包含有 Expanding 窗口部件的窗体的尺寸大小时,多出来的空间就会分配给 Expanding 窗口部件,而 Preferred 窗口部件仍旧会按照原有大小提示而保持不变。

这里还有另外两种大小规则:MinimumExpanding 和 Ignored。前者仅仅用于 Qt 老版本的极少数情况中,但是如今它已经不再会被用到了。一种比较好的方式是使用 Expanding,并且再适当地对 minimumSizeHint()进行重新实现即可。后者与 Expanding 相似,只是它可以忽略窗口部件的大小提示和最小大小提示。

除了大小规则中包含的水平方向和垂直方向两个分量之外，QSizePolicy类还保存了水平方向和垂直方向的一个拉伸因子。这些拉伸因子可以用来说明在增大窗体时，对不同的子窗口部件应使用的不同放大比例。例如，假定在一个 QTextEdit 的上面还有一个 QTreeWidget，并且希望这个 QTextEdit 的高度能够是 QTreeWidget 高度的两倍，那么就可以把这个 QTextEdit 在垂直方向上的拉伸因子设置为 2，而把 QTreeWidget 在垂直方向上的拉伸因子设置为 1。

影响布局方式的另一种方法是设置它的子窗口部件的最小大小、最大大小或固定大小。当布局管理器在摆放这些窗口部件的时候，它就会考虑这些约束条件。并且如果这样还不够的话，还可以对子窗口部件的类进行派生并且重新实现 sizeHint()函数，由此获得所需的大小提示。

6.2 分组布局

QStackedLayout类可以对一组子窗口部件进行摆放，或者对它们进行“分页”，而且一次只显示其中一个，而把其他的子窗口部件或者分页都隐藏起来。QStackedLayout 本身并不可见，并且对于用户改变分页也没有提供其他特有的方法。图 6.5 中的小箭头和暗灰色框架是由 Qt 设计师提供的，利用它们可以使布局设计变得更容易些。为方便起见，Qt 还提供了 QStackedWidget 类，这个类提供了一个带内置 QStackedLayout 的 QWidget。

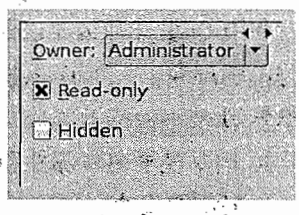


图 6.5 QStackedLayout

分页是从 0 开始编号的。要使某个特定的子窗口部件可见，可以用一个页号来调用 setCurrentIndex()。使用 indexOf()可以获取子窗口部件的页号。

在图 6.6 中显示的 Preferences 对话框就是一个使用了 QStackedLayout 的例子。这个对话框由左侧的 QListWidget 和右侧的 QStackedLayout 构成。在 QListWidget 中的每一项，都分别对应于 QStackedLayout 中的不同页。这里给出了与这个对话框的构造函数相关的部分代码：

```
PreferenceDialog::PreferenceDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    listWidget = new QListWidget;
    listWidget->addItem(tr("Appearance"));
    listWidget->addItem(tr("Web Browser"));
    listWidget->addItem(tr("Mail & News"));
    listWidget->addItem(tr("Advanced"));

    stackedLayout = new QStackedLayout;
    stackedLayout->addWidget(appearancePage);
    stackedLayout->addWidget(webBrowserPage);
    stackedLayout->addWidget(mailAndNewsPage);
    stackedLayout->addWidget(advancedPage);
    connect(listWidget, SIGNAL(currentRowChanged(int)),
            stackedLayout, SLOT(setCurrentIndex(int)));
    ...
    listWidget->setCurrentRow(0);
}
```

我们创建一个 QListWidget，并且把它和这些分页的名字一起配合使用。然后，创建一个 QStackedLayout，并且对每一个分页分别调用 addWidget()。我们把这个列表窗口部件的 currentRowChanged(int)信号连接到这个分组布局的 setCurrentIndex(int)，可以用来实现分页的切换操作，并且在构造函数的最后，对这个列表窗口部件调用 setCurrentRow()，这样就可以把页号 0 设置为起始页。

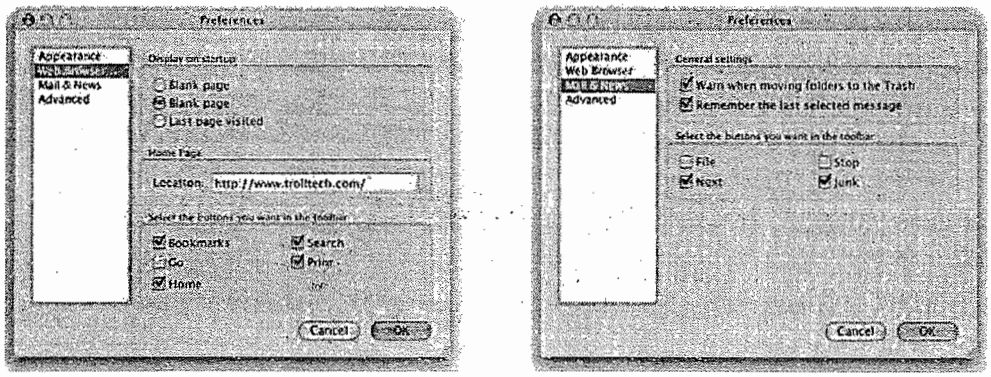


图 6.6 Preferences 对话框中的两个分页

利用 Qt 设计师创建这样的对话框也是非常容易的:

1. 基于"Dialog"或者"Widget"模板,创建一个新的窗体。
2. 在这个窗体上添加一个 QListWidget 和一个 QStackedWidget。
3. 用一些子窗口部件和布局来填充每一个分页。
 (要创建一个新的分页,可以在窗体上单击鼠标右键并且选择"Insert Page";如果要在不同的分页间切换,可以点击 QStackedWidget 右上角的那个向左或者向右的小箭头。)
4. 使用水平布局,把这些窗口部件一个挨一个地摆放好。
5. 把这个列表窗口部件的 currentRowChanged(int)信号与分组窗口部件的 setCurrentIndex(int)槽连接起来。
6. 把列表框的 currentRow 属性值设置为 0。

由于已经使用预先定义的信号和槽来实现了分页之间的切换操作,所以当我们在 Qt 设计师中预览这个对话框时,它将可以直接显示出正确的行为特性。

对于页数较少或者可能会保持较小的一些情况,一种比使用 QStackedWidget 和 QListWidget 更为简单的替换方法是使用 QTabWidget。

6.3 切分窗口

QSplitter 就是一个可以包含一些其他窗口部件的窗口部件。在切分窗口(splitter)中的这些窗口部件会通过切分条(splitter handle)而分隔开来。用户可以通过拖动这些切分条来改变切分窗口中子窗口部件的大小。切分窗口常常可以用作布局管理器的替代品,从而可以把更多的控制权交给用户。

QSplitter 中的子窗口部件将会自动按照创建时的顺序一个挨一个地(或者一个在另外一个的下面)放在一起,并以切分窗口拖动条(splitter bar)来分隔相邻窗口部件。以下是用于创建如图 6.7 所示窗口的代码。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QTextEdit *editor1 = new QTextEdit;
    QTextEdit *editor2 = new QTextEdit;
    QTextEdit *editor3 = new QTextEdit;

    QSplitter splitter(Qt::Horizontal);
```

```
    splitter.addWidget(editor1);
    splitter.addWidget(editor2);
    splitter.addWidget(editor3);
    ...
    splitter.show();
    return app.exec();
}
```

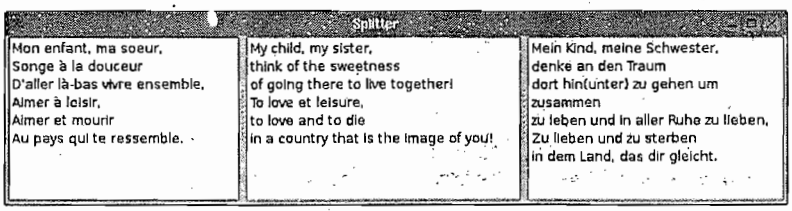


图 6.7 Splitter 应用程序

这个例子由三个 QTextEdit 组成，通过一个 QSplitter 窗口部件水平地摆放它们，图 6.8 给出了它们的构成原理图。与布局管理器不同之处在于，布局管理器只是简单地摆放一个窗体中的子窗口部件并且也没有可见的外形，但 QSplitter 是从 QWidget 派生的，并且在使用的时候，它也可以像任何其他窗口部件一样使用。

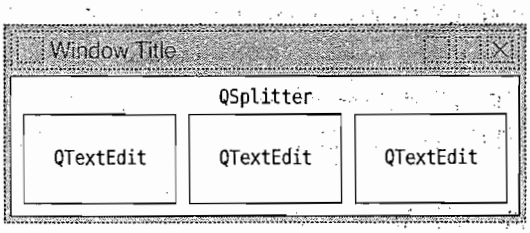


图 6.8 Splitter 应用程序的窗口部件

通过对多个 QSplitter 进行水平或者垂直方向的嵌套，就可以获得更为复杂的一些布局。例如，在如图 6.9 所示的 Mail Client(邮件客户端)应用程序中，就由一个水平的 QSplitter 构成，而在它的右侧又包含了一个垂直的 QSplitter。图 6.10 给出了该布局的示意图。

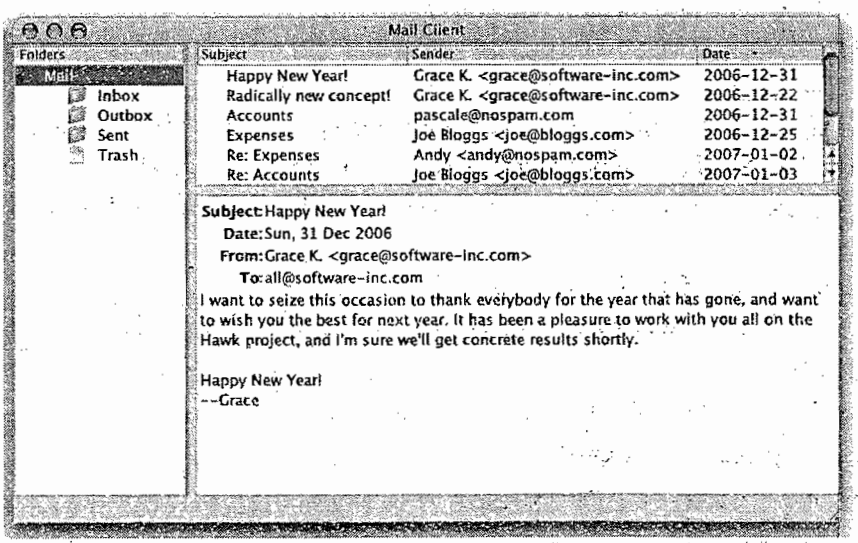


图 6.9 Mail Client 应用程序

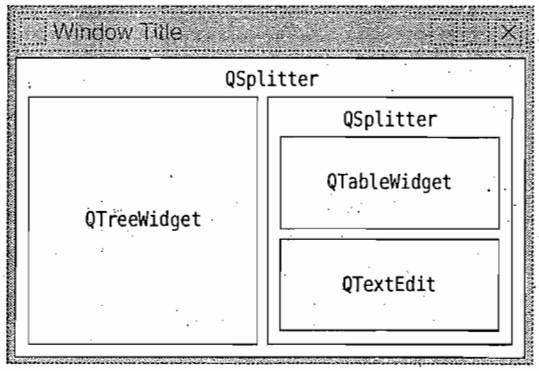


图 6.10 Mail Client 的切分窗口布局图

以下是 Mail Client 应用程序 QMainWindow 子类的构造函数中所使用的代码:

```
MailClient::MailClient()
{
    ...
    rightSplitter = new QSplitter(Qt::Vertical);
    rightSplitter->addWidget(messagesTreeWidget);
    rightSplitter->addWidget(textEdit);
    rightSplitter->setStretchFactor(1, 1);

    mainSplitter = new QSplitter(Qt::Horizontal);
    mainSplitter->addWidget(foldersTreeWidget);
    mainSplitter->addWidget(rightSplitter);
    mainSplitter->setStretchFactor(1, 1);
    setCentralWidget(mainSplitter);

    setWindowTitle(tr("Mail Client"));
    readSettings();
}
```

在创建了我们希望显示的三个窗口部件之后,再创建一个垂直切分窗口 rightSplitter,并且把我们想要的两个窗口部件添加到它的右侧。然后,再创建一个水平切分窗口 mainSplitter,并且把我们想要在左侧显示的窗口部件和想要在右侧显示的 rightSplitter 等这些窗口部件统统放进去。我们把 mainSplitter 作为 QMainWindow 的中央窗口部件。

当用户重新改变窗口的大小时,QSplitter 通常会重新分配空间,以便能够使所有的子窗口部件的相对大小能够与先前一样保持相同的比例。在 Mail Client 例子中,我们想要的不是这种行为,相反,我们希望 QTreeWidget 和 QTableWidget 保持它们的大小不变,而把任何多余的额外空间都留给 QTextEdit。这一点可以通过两次调用 setStretchFactor()来实现。第一个参数是切分窗口子窗口部件的索引值,该值是一个从 0 开始的整数值;第二个参数是我们想要设置的伸展因子,伸展因子的默认值是 0。

第一次是对 rightSplitter 调用 setStretchFactor(),它会把位置 1 处的窗口部件(textEdit)的伸展因子设置为 1。第二次则是对 mainSplitter 调用 setStretchFactor(),它会把位置 1 处的窗口部件(rightSplitter)的伸展因子设置为 1。这样将可以确保 textEdit 总是可以获取那些任何多余的可用空间。

在启动应用程序时,QSplitter 会根据子窗口部件的初始大小(或者在没有给定初始大小的时候,根据它们的大小提示)给它们分配合适的大小。在程序中,可以通过调用QSplitter::setSizes()来移动切分条。QSplitter 类也可以保存它的状态,并且可以在下次运行应用程序的时候直接恢复它的状态值。以下给出了 Mail Client 应用程序的 writeSettings()函数,它可以保存 Mail Client 的各个状态设置值:

```
void MailClient::writeSettings()
{
    QSettings settings("Software Inc.", "Mail Client");

    settings.beginGroup("mainWindow");
    settings.setValue("geometry", saveGeometry());
    settings.setValue("mainSplitter", mainSplitter->saveState());
    settings.setValue("rightSplitter", rightSplitter->saveState());
    settings.endGroup();
}
```

以下是用于相应的 readSettings()函数中的代码：

```
void MailClient::readSettings()
{
    QSettings settings("Software Inc.", "Mail Client");

    settings.beginGroup("mainWindow");
    restoreGeometry(settings.value("geometry").toByteArray());
    mainSplitter->restoreState(
            settings.value("mainSplitter").toByteArray());
    rightSplitter->restoreState(
            settings.value("rightSplitter").toByteArray());
    settings.endGroup();
}
```

Qt 设计师完全支持 QSplitter。要把多个窗口部件放到一个切分窗口中，可以先把这些子窗口部件放置在期望的大致位置，选中它们，然后再单击 Form→Lay Out Horizontally in Splitter 或者 Form→Lay Out Vertically in Splitter 即可。

6.4 滚动区域

QScrollArea 类提供了一个可以滚动的视口和两个滚动条。如果想给一个窗口部件添加一个滚动条，则可以使用一个 QScrollArea 类来实现，这可能要比我们自己通过初始化 QScrollBar，然后再实现它的滚动等功能简单得多。

QScrollArea 的使用方法，就是以我们想要添加滚动条的窗口部件为参数调用 setWidget()。如果这个窗口部件的父对象不是视口，QScrollArea 会自动把这个窗口部件的父对象重定义为该视口(可以通过 QScrollArea::viewport()来访问)，并且让它成为视口的子对象。例如，如果想在第 5 章中开发的 IconEditor 窗口部件的周围添加滚动条(如图 6.11 所示)，则可以编写如下代码：

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    IconEditor *iconEditor = new IconEditor;
    iconEditor->setIconImage(QImage(":/images/mouse.png"));

    QScrollArea scrollArea;
    scrollArea.setWidget(iconEditor);
    scrollArea.viewport()->setBackgroundRole(QPalette::Dark);
    scrollArea.viewport()->setAutoFillBackground(true);
    scrollArea.setWindowTitle(QObject::tr("Icon Editor"));

    scrollArea.show();
    return app.exec();
}
```

图 6.12 给出了 QScrollArea 的原理图，它会以窗口部件的当前大小来显示它，或者在没有重新改变窗口部件大小的时候以它的大小提示来显示它。通过调用 setWidgetResizable(true)，可以告 QScrollArea 要自动重新改变该窗口部件的大小，以利用超过它的大小提示之外的任何多余空间

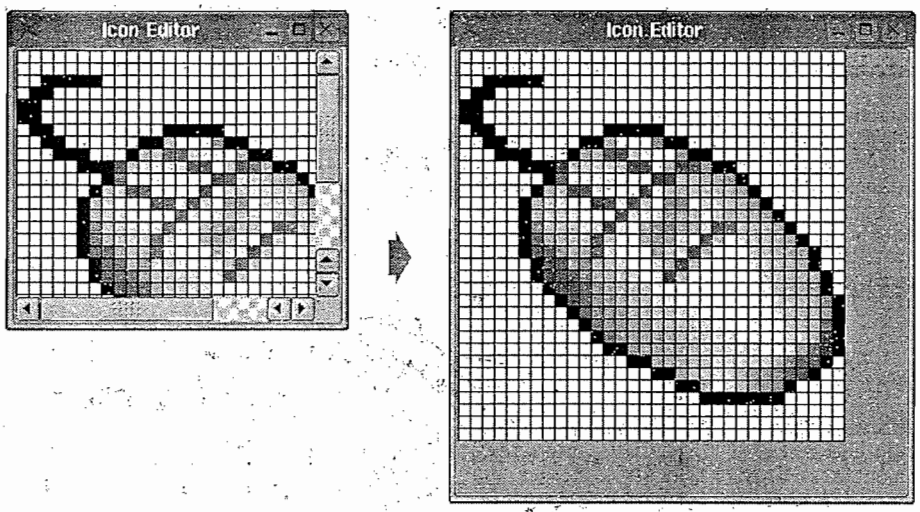


图 6.11 重新调整 QScrollArea 的大小

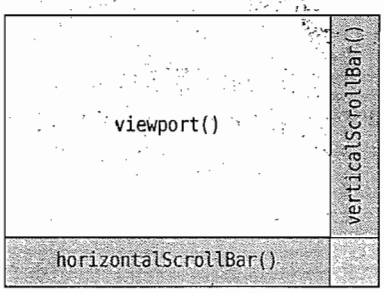


图 6.12 QScrollArea 的窗口部件构成

默认情况下,只有在视口的大小小于子窗口部件的大小时,才会把滚动条显示出来。但通过设置滚动条的策略,可以强制滚动条总是可见:

```
scrollArea.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
scrollArea.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
```

QScrollArea 从 QAbstractScrollArea 继承了它的许多功能。像 QTextEdit 和 QAbstractItemView(Qt 项视图类的基类)这样的一些类,由于它们是从 QAbstractScrollArea 中派生出来的,所以为了获得滚动条,就没有必要再把它们封装在 QScrollArea 中。

6.5 停靠窗口和工具栏

停靠窗口(dock window)是指一些可以停靠在 QMainWindow 中或是浮动为独立窗口的窗口。QMainWindow 提供了 4 个停靠窗口区域:分别在中央窗口部件的上部、下部、左侧和右侧。诸如像 Microsoft Visual Studio 和 Qt Linguist 这样的应用程序都广泛使用了停靠窗口,以提供一种非常灵活的用户接口方式。在 Qt 中,各个停靠窗口都是 QDockWidget 的实例。图 6.13 给出了一个带有工具栏和停靠窗口的 Qt 应用程序。

每一个停靠窗口都有自己的标题栏,即使它处于停靠时也是如此。通过拖拽这一标题栏,用户可以把停靠窗口从一个停靠区域移动到另外一个停靠区域。通过把这个停靠窗口拖动到其他停靠区域的外面,就可以把停靠窗口从一个停靠区域中分离出来,让它成为一个独立的窗口。自

由浮动的停靠窗口总是显示在它们的主窗口的上面。通过点击窗口部件标题栏上的“关闭”按钮，就可以关闭 QDockWidget。通过调用 QDockWidget::setFeatures()，就可以禁用所有这些特性以及它们的任意组合。

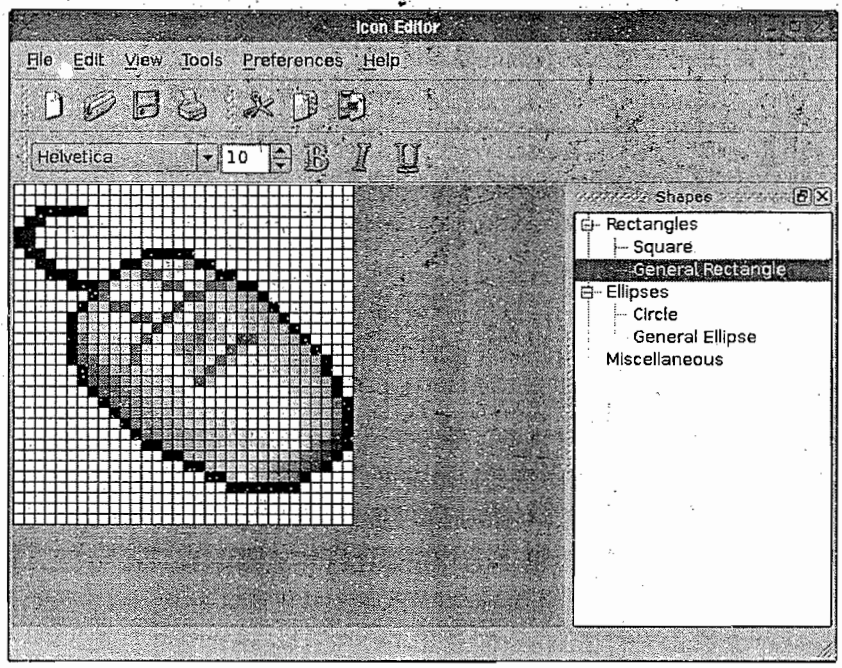


图 6.13 带一个停靠窗口的 QMainWindow

在 Qt 的早期版本中，工具栏采用与停靠窗口一样的处理方式，并且共享同一停靠区域。从 Qt 4开始，工具栏围绕中央窗口部件，占有它们自己的区域(如图 6.14 所示)，并且不能取消停靠(undock)。如果需要一个浮动工具栏，只需把它放进 QDockWidget 即可。

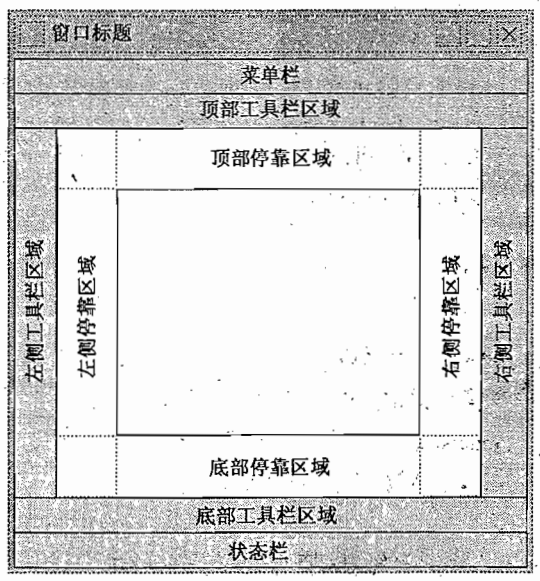


图 6.14 QMainWindow 的停靠区域和工具栏区域

用虚线显示的四个角可以属于两个相邻停靠区域中的任何一个。例如,假定我们需要让左上角属于左侧的停靠区域,则只需调用 QMainWindow::setCorner(Qt::TopLeftCorner, Qt::LeftDockWidgetArea)即可。

以下程序片段说明了如何对 QDockWidget 中已经存在的窗口部件(在这个例子中,就是一个 QTreeWidget)进行封装,并把它插入到右侧的停靠区域:

```
QDockWidget *shapesDockWidget = new QDockWidget(tr("Shapes"));
shapesDockWidget->setObjectName("shapesDockWidget");
shapesDockWidget->setWidget(treeWidget);
shapesDockWidget->setAllowedAreas(Qt::LeftDockWidgetArea
                                  | Qt::RightDockWidgetArea);
addDockWidget(Qt::RightDockWidgetArea, shapesDockWidget);
```

setAllowedAreas()调用说明对停靠区域加以限定即可以接受停靠窗口。在此给出的代码中,只允许把停靠窗口拖拽到左侧和右侧的停靠区域,这两个地方都有显示它的足够垂直空间,因而可以合理地把它显示出来。如果没有明确地设置所允许的区域,那么用户就可能把该停靠窗口拖动到这四个可停靠区域中的任何一个地方。

每个 QObject 都可以给定一个"对象名"。在进行程序调试时,这个名字会非常有用,并且一些测试工具也会用到它。通常,我们不必费劲地给定窗口部件的名字,但是在创建一些停靠窗口和工具栏时,如果希望使用 QMainWindow::saveState()和 QMainWindow::restoreState()来保存、恢复停靠窗口和工具栏的几何形状及状态的话,给定窗口部件的名字就很有必要了。

下面的代码显示了如何创建一个工具栏的过程,该工具栏包含一个 QComboBox、一个 QSpinBox 和一些 QToolButton。其中的这些 QToolButton 来自 QMainWindow 子类的构造函数:

```
QToolBar *fontToolBar = new QToolBar(tr("Font"));
fontToolBar->setObjectName("fontToolBar");
fontToolBar->addWidget(familyComboBox);
fontToolBar->addWidget(sizeSpinBox);
fontToolBar->addAction(boldAction);
fontToolBar->addAction(italicAction);
fontToolBar->addAction(underlineAction);
fontToolBar->setAllowedAreas(Qt::TopToolBarArea
                             | Qt::BottomToolBarArea);
addToolBar(fontToolBar);
```

如果想保存所有停靠窗口和工具栏的位置,以使下一次运行应用程序时能够恢复它们的值,那么可以像以前在保存一个 QSplitter 的状态值时所使用的代码一样来编写类似的代码,也就是使用 QMainWindow 的 saveState()和 restoreState()函数:

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");

    settings.beginGroup("mainWindow");
    settings.setValue("geometry", saveGeometry());
    settings.setValue("state", saveState());
    settings.endGroup();
}

void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");

    settings.beginGroup("mainWindow");
    restoreGeometry(settings.value("geometry").toByteArray());
    restoreState(settings.value("state").toByteArray());
    settings.endGroup();
}
```

最后，QMainWindow 提供了一个上下文菜单，其中列出了所有的停靠窗口和工具栏。图 6.15 给出了这个上下文菜单。用户可以使用这个菜单关闭和恢复停靠窗口，也可以用它隐藏和恢复工具栏。

6.6 多文档界面

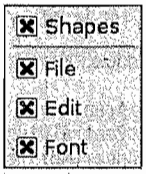


图 6.15 QMainWindow 的上下文菜单

在主窗口的中央区域能够提供多个文档的那些应用程序就称为多文档界面(Multiple Document Interface，MDI)应用程序，或者称为 MDI 应用程序。在 Qt 中，通过把 QMdiArea 类作为中央窗口部件，并且通过让每一个文档窗口都成为这个 QMdiArea 的子窗口部件，就可以创建一个多文档界面应用程序了。

对于多文档界面应用程序有一个惯例，就是为它提供一个 Window 菜单，这个菜单中包含一些管理这些窗口以及这些窗口列表的命令。激活窗口会使用一个选择标记标识出来。用户通过在 Window 菜单中单击代表特定窗口的一项，就可以激活任何窗口。

这一节将开发如图 6.16 所示的 Editor 多文档界面应用程序，从而说明如何创建一个多文档界面应用程序以及如何实现它的 Window 菜单。该应用程序的全部菜单如图 6.17 所示。

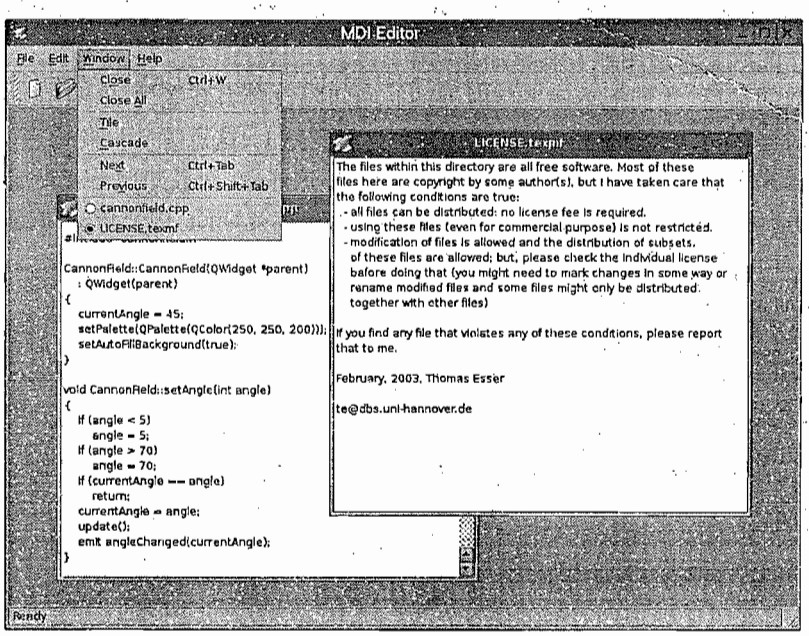


图 6.16 Editor 多文档界面应用程序

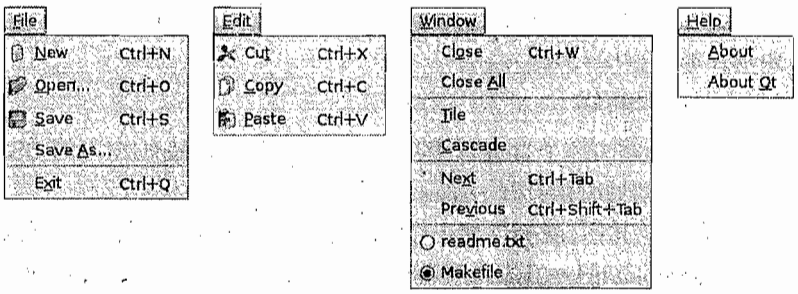


图 6.17 Editor 多文档界面应用程序的菜单

这个应用程序由两个类组成:MainWindow 类和 Editor 类。在本书的例子程序中提供了它的代码,并且由于其中的绝大部分代码都与第一部分中的 Spreadsheet 应用程序相同或者相似,所以我们仅仅给出那些与多文档界面相关的代码。

让我们先从 MainWindow 类开始:

```
MainWindow::MainWindow()
{
    mdiArea = new QMdiArea;
    setCentralWidget(mdiArea);
    connect(mdiArea, SIGNAL(subWindowActivated(QMdiSubWindow*)),
            this, SLOT(updateActions()));

    createActions();
    createMenus();
    createToolBars();
    createStatusBar();

    setWindowIcon(QPixmap(":/images/icon.png"));
    setWindowTitle(tr("MDI Editor"));
    QTimer::singleShot(0, this, SLOT(loadFiles()));
}
```

在 MainWindow 构造函数中,创建了一个 QMdiArea 窗口部件,并且让它成为中央窗口部件。我们把 QMdiArea 的 subWindowActivated()信号与将要用来保持更新 Window 菜单的槽连接起来,并且会根据应用程序的状态来启用或者禁用那些动作。

在构造函数的最后,我们把单触发定时器的时间间隔设置为 0 毫秒,以调用 loadFiles()函数。对于这样的定时器,只要事件循环一空闲就会触发它。实际上,这意味着只要构造函数结束,同时在主窗口显示出来之后,它就会调用 loadFiles()。如果不这样做,而且如果还需要加载许多大文件的话,那么该构造函数在文件加载完毕之前就无法结束。在此期间,用户极有可能在屏幕上看不到任何东西,这样他可能会认为是应用程序启动失败了。

```
void MainWindow::loadFiles()
{
    QStringList args = QApplication::arguments();
    args.removeFirst();
    if (!args.isEmpty()) {
        foreach (QString arg, args)
            openFile(arg);
        mdiArea->cascadeSubWindows();
    } else {
        newFile();
    }
    mdiArea->activateNextSubWindow();
}
```

如果用户在命令行中启动该应用程序时使用了一个或者多个文件名,那么这个函数就会试图加载每一个文件,并且会按照逐级层叠的方式显示这些子窗口,以便用户可以轻松地看到它们。Qt 的一些特殊命令行选项,比如-style 和-font,QApplication 的构造函数会自动把它们从参数列表中剔除出去。因此,如果在命令行中写成:

```
mdieditor -style motif readme.txt
```

QApplication::arguments()就会返回一个含有两个项("mdieditor"和"readme.txt")的 QStringList,那么 MDIEditor 应用程序就会在启动的时候打开 readme.txt。

如果在命令行中没有指定文件,就会创建一个空的编辑器子窗口,以便用户可以方便地直接开始输入。对于 activateNextSubWindow()的调用意味着对编辑器窗口赋予焦点,并且可以确保调用 updateActions()函数来更新 Window 菜单,以及根据应用程序的状态来启用和禁用一些动作。

```
void MainWindow::newFile()
{
    Editor *editor = new Editor;
    editor->newFile();
    addEditor(editor);
}
```

newFile()槽对 File→New 菜单项做出响应。它创建一个 Editor 窗口部件,并且把它传递给 addEditor()私有函数。

```
void MainWindow::open()
{
    Editor *editor = Editor::open(this);
    if (editor)
    addEditor(editor);
}
```

open()函数对 File→Open 菜单项做出响应。它调用 Editor::open()函数,该函数会弹出一个文件对话框。如果用户选择了一个文件,就会创建一个新的 Editor,读入文件的文本内容,并且如果读取成功,就会返回一个指向 Editor 的指针。如果用户取消了该文件对话框,或者如果读取失败,就会返回一个空指针并通知用户出错。在 Editor 类中实现文件操作要比在 MainWindow 类中实现文件操作更有意义,因为每个 Editor 都需要维护它自己的独立状态。

```
void MainWindow::addEditor(Editor *editor)
{
    connect(editor, SIGNAL(copyAvailable(bool)),
            cutAction, SLOT(setEnabled(bool)));
    connect(editor, SIGNAL(copyAvailable(bool)),
            copyAction, SLOT(setEnabled(bool)));

    QMdiSubWindow *subWindow = mdiArea->addSubWindow(editor);
    windowMenu->addAction(editor->windowMenuAction());
    windowActionGroup->addAction(editor->windowMenuAction());
    subWindow->show();
}
```

addEditor()私有函数会从 newFile() 和 open()中得到调用,以完成一个新的 Editor 窗口部件的初始化。它以创建两个信号-槽的连接开始。根据是否有选中的任意文本,这两个连接可以确保 Edit→Cut 和 Edit→Copy 菜单的启用或者禁用。

因为我们正在使用的是一个多文档界面,所以完全有可能同时使用多个 Editor 窗口部件。这一点之所以需要关注,是因为我们只对来自激活的 Editor 窗口的 copyAvailable(bool)信号做出响应,而不会对其他的 Editor 窗口做出响应。但由于只能由激活的窗口发射这些信号,所以,实际上这并不算什么问题。

QMdiArea::addSubWindow()函数创建一个新的 QMdiSubWindow,把作为参数传递的该窗口部件放进子窗口中,并且返回该子窗口。接下来,我们创建一个 QAction,由其代表 Window 菜单中的一个窗口。该动作由 Editor 类提供,后面会说明这个类。我们还会在一个 QActionGroup 对象中添加该动作。QActionGroup 对象可以确保每个时刻只能选中 Window 菜单项中的一项。最后,对新的 QMdiSubWindow 调用 show(),使其可见。

```
void MainWindow::save()
{
    if (activeEditor())
        activeEditor()->save();
}
```

如果存在激活的编辑器,save()槽就会对它调用 Editor::save()。同样,执行真正工作的代码就放在 Editor 类中。

```
Editor *MainWindow::activeEditor()
{
    QMdiSubWindow *subWindow = mdiArea->activeSubWindow();
    if (subWindow)
        return qobject_cast<Editor *>(subWindow->widget());
    return 0;
}
```

activeEditor()私有函数返回一个类型为 Editor 的指针,它指向当前激活的子窗口;或者在没有激活的子窗口时,返回一个空指针。

```
void MainWindow::cut()
{
    if (activeEditor())
        activeEditor()->cut();
}
```

cut()槽可以对激活的编辑器调用 Editor::cut()。由于 copy()槽和 paste()槽都与 cut()槽遵循相同的模式,所以没有再给出它们的代码。

```
void MainWindow::updateActions()
{
    bool hasEditor = (activeEditor() != 0);
    bool hasSelection = activeEditor()
                        && activeEditor()->textCursor().hasSelection();

    saveAction->setEnabled(hasEditor);
    saveAsAction->setEnabled(hasEditor);
    cutAction->setEnabled(hasSelection);
    copyAction->setEnabled(hasSelection);
    pasteAction->setEnabled(hasEditor);
    closeAction->setEnabled(hasEditor);
    closeAllAction->setEnabled(hasEditor);
    tileAction->setEnabled(hasEditor);
    cascadeAction->setEnabled(hasEditor);
    nextAction->setEnabled(hasEditor);
    previousAction->setEnabled(hasEditor);
    separatorAction->setVisible(hasEditor);

    if (activeEditor())
        activeEditor()->windowMenuAction()->setChecked(true);
}
```

每当一个新的子窗口成为激活窗口,或者在关闭最后一个子窗口时,都会发射 subWindowActivated()信号。在后一种情况下,它的参数就是一个空指针。这个信号会连接到 updateActions()槽。

只有在存在激活窗口时,绝大多数的菜单项才会起作用。如果不存在激活窗口,就可以禁用它们。最后,我们在 QAction 上调用 setChecked(),表示这是一个激活窗口。由于有了 QActionGroup 的帮助,就不必再明确地对前面的激活窗口进行解除选定操作了。

```
void MainWindow::createMenus()
{
    ...
    windowMenu = menuBar()->addMenu(tr("&Window"));
    windowMenu->addAction(closeAction);
    windowMenu->addAction(closeAllAction);
    windowMenu->addSeparator();
    windowMenu->addAction(tileAction);
    windowMenu->addAction(cascadeAction);
    windowMenu->addSeparator();
    windowMenu->addAction(nextAction);
    windowMenu->addAction(previousAction);
    windowMenu->addAction(separatorAction);
    ...
}
```

createMenus()私有函数会让这些动作和 Window 菜单一起配合工作。所有这些动作都是一些这种典型的菜单项,并且可以很容易地使用 QMdiArea 的 closeActiveSubWindow()、closeAllSubWindows()、tileSubWindows()和 cascadeSubWindows()槽来实现它们。每当打开一个新窗口时,都会把该动作添加到 Window 菜单的动作列表中。[在 123 页的 addEditor()函数中,就已经看到了是如何完成这一过程的。]当用户关闭编辑器窗口时,就会删除它在 Window 菜单中的对应动作(因为该编辑器窗口拥有这个动作),因而就会自动从 Window 菜单中移除这个动作。

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    mdiArea->closeAllSubWindows();
    if (!mdiArea->subWindowList().isEmpty()) {
        event->ignore();
    } else {
        event->accept();
    }
}
```

重新实现 closeEvent()函数来关闭所有的子窗口,从而使得每个子窗口都要接收一个关闭事件。如果这些子窗口的其中之一"忽略"了它的关闭事件(可能是因为用户撤销了一个"未保存变化"的消息框),那么也就忽略对 MainWindow 的关闭事件;否则,我们就接受它,这样的结果就是 Qt 会关闭整个应用程序。如果没有在 MainWindow 中重新实现它的 closeEvent(),那么将不会给用户留下对那些"未保存变化"进行存储的任何机会。

现在已经完成了对 MainWindow 窗口的查看,因而就可以转到对 Editor 的实现上。Editor 类表示一个子窗口,它继承了 QTextEdit,而 QTextEdit 提供了文本编辑功能。在实际的应用程序中,如果需要一个代码编辑的组件,也许会考虑使用 Scintilla,可以从 http://www.riverbankcomputing.co.uk/qscintilla/中获取专门用于 Qt 的组件 QScintilla。

就像任何 Qt 窗口部件都可以作为一个单独的窗口一样,任何 Qt 窗口部件都可以放进 QMdiSubWindow 中,并将其作为多文档界面窗口工作区中的一个子窗口。

以下是这个 Editor 类的定义:

```
class Editor : public QTextEdit
{
    Q_OBJECT

public:
    Editor(QWidget *parent = 0);

    void newFile();
    bool save();
    bool saveAs();
    QSize sizeHint() const;
    QAction *windowMenuAction() const { return action; }

    static Editor *open(QWidget *parent = 0);
    static Editor *openFile(const QString &fileName,
                            QWidget *parent = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void documentWasModified();

private:
    bool okToContinue();
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
```

```
    QString strippedName(const QString &fullFileName);

    QString curFile;
    bool isUntitled;
    QAction *action;
};
```

Editor 类中已经包含了在 Spreadsheet 应用程序的 MainWindow 类中出现过的 4 个私有函数:okToContinue()、saveFile()、setCurrentFile()和 strippedName()。

```
Editor::Editor(QWidget *parent)
    : QTextEdit(parent)
{
    action = new QAction(this);
    action->setCheckable(true);
    connect(action, SIGNAL(triggered()), this, SLOT(show()));
    connect(action, SIGNAL(triggered()), this, SLOT(setFocus()));

    isUntitled = true;

    connect(document(), SIGNAL(contentsChanged()),
            this, SLOT(documentWasModified()));

    setWindowIcon(QPixmap(":/images/document.png"));
    setWindowTitle("[*]");
    setAttribute(Qt::WA_DeleteOnClose);
}
```

首先,创建一个 QAction,用它表示应用程序 Window 菜单中的编辑器,并且把这个动作与 show()和 setFocus()槽连接起来。

由于允许用户创建任意数量的编辑器窗口,所以必须为它们预备一些名字,这样就可以在第一次保存这些窗口之前把它们区分开来。处理这种情况的一种常用方式是分配一个包含数字的名称(例如,document1.txt)。我们使用一个 isUntitled 变量来区分是用户提供的名称还是程序自动创建的名称。

我们把文本文档的 contentsChanged()信号连接到 documentWasModified()私有槽上。这个槽只会简单地调用 setWindowModified(true)。

最后,为了防止用户关闭 Editor 窗口时出现内存泄漏,需要设置 Qt::WA_DeleteOnClose 属性。

```
void Editor::newFile()
{
    static int documentNumber = 1;

    curFile = tr("document%1.txt").arg(documentNumber);
    setWindowTitle(curFile + "[*]");
    action->setText(curFile);
    isUntitled = true;
    ++documentNumber;
}
```

newFile()函数会为新的文档产生一个像 document1.txt 这样的名称。由于当在一个新创建的 Editor 中调用 open()而打开一个已经存在的文档时,我们并不想白白浪费一个数字,因而把这段代码放在了 newFile()中,而不是放在构造函数中。由于 documentNumber 声明为静态变量,所以它可以被所有的 Editor 实例共享。

在窗口标题中的"[*]"标记是一种位置标记符,在除 Mac OS X 系统平台之外的文件中,当存在未保存的变化时,我们就希望能够让这个星号出现。而在 Mac OS X 系统中,未保存的文档会在它们的窗口关闭按钮中出现一个点。已经在第 3 章(见 48 页)说明过这个位置标记符的用法。

除了创建一些新文件之外,用户通常希望打开一些已经存在的文件,即从文件对话框中打开已有文件,或者从文件列表中打开已有文件。比如,从最近打开文件列表中选择一些文件打开。

为了支持这些用法，Qt 提供了两个静态函数：open()用于从文件系统中选择一个文件的名字，而 openFile()用于创建一个 Editor 并且读入一个指定文件的内容。

```
Editor *Editor::open(QWidget *parent)
{
    QString fileName =
            QFileDialog::getOpenFileName(parent, tr("Open"), ".");
    if (fileName.isEmpty())
        return 0;

    return openFile(fileName, parent);
}
```

静态函数 open()会弹出一个文件对话框，用户可以通过它选择一个文件。如果选择了文件，就会调用 openFile()创建一个 Editor 并且读入该文件的内容。

```
Editor *Editor::openFile(const QString &fileName, QWidget *parent)
{
    Editor *editor = new Editor(parent);
    if (editor->readFile(fileName)) {
        editor->setCurrentFile(fileName);
        return editor;
    } else {
        delete editor;
        return 0;
    }
}
```

这个 openFile()静态函数从创建新的 Editor 窗口部件开始，然后会试着读入指定的文件。如果读入成功，就会返回该 Editor；否则，用户就会得到出现问题的信息[在 readFile()中]，删除该编辑器，并且返回一个空指针。

```
bool Editor::save()
{
    if (isUntitled) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}
```

save()函数使用 isUntitled 变量来决定它是应该调用 saveFile()还是应该调用 saveAs()。

```
void Editor::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        event->accept();
    } else {
        event->ignore();
    }
}
```

这里重新实现了 closeEvent()函数，它允许用户保存那些未保存的变化。这部分逻辑判断代码放在 okToContinue()函数中，该函数可以弹出一个消息框，询问“Do you want to save your changes?”(你想保存你的改变吗?)。如果 okToContinue()函数返回 true，就接受这个关闭事件；否则，就“忽略”它，并且让这个窗口不受其影响。

```
void Editor::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    isUntitled = false;
    action->setText(strippedName(curFile));
    document()->setModified(false);
    setWindowTitle(strippedName(curFile) + "[*]");
```

```
    setWindowModified(false);
}
```

setCurrentFile()函数会在 openFile()和 saveFile()中得到调用,用来更新 curFile 和 isUntitled 变量,设置窗口的标题和动作的文本,以及用来把这个文档的"modified"标记设置成 false。一旦用户修改了编辑器中的文本,底层的 QTextDocument 就会发射 contentsChanged()信号,并且把它内部的"modified"标记设置成 true。

```
QSize Editor::sizeHint() const
{
    return QSize(72 * fontMetrics().width('x'),
                 25 * fontMetrics().lineSpacing());
}
```

最后,根据字符"x"的宽度和文本行的高度,sizeHint()函数可以返回一个大小提示。QMdiArea 可以使用这个大小提示为窗口设定一个初始的尺寸大小值。

多文档界面就是同时处理多个文档的一种方法。在 Mac OS X 系统中,一种更好的方法是使用多个顶层窗口。第 3 章的 3.7 节中已经介绍了这种方法。

第7章 事件处理

事件(event)是由窗口系统或者 Qt 自身产生的,用以响应所发生的各类事情。当用户按下或者松开键盘或者鼠标上的按键时,就可以产生一个键盘或者鼠标事件;当某个窗口第一次显示的时候,就会产生一个绘制事件,用来告知窗口需要重新绘制它本身,从而使得该窗口可见。大多数事件是作为用户动作的响应而产生的,但是也有一些例外,比如像定时器事件,则是由系统独立产生的。

在使用 Qt 进行编程开发时,基本不需要考虑事件,因为在发生某些重要的事情时,Qt 窗口部件都会发射信号。但是当我们需要编写自己的自定义窗口部件,或者是当我们希望改变已经存在的 Qt 窗口部件的行为时,事件就变得非常有用了。

不应该混淆"事件"和"信号"这两个概念。一般情况下,在使用窗口部件的时候,信号是十分有用的;而在实现窗口部件时,事件则是十分有用的。例如,当使用 QPushButton 时,我们对于它的 clicked()信号往往更为关注,而很少关心促成发射该信号的底层鼠标或者键盘事件。但是,如果要实现的是一个类似于 QPushButton 的类,就需要编写一定的处理鼠标和键盘事件的代码,而且在必要的时候,还需要发射 clicked()信号。

7.1 重新实现事件处理器

在 Qt 中,事件就是 QEvent 子类的一个实例。Qt 处理的事件类型有一百多种,其中的每一种都可以通过一个枚举值来进行识别。例如,QEvent::type()可以返回用于处理鼠标按键事件的 QEvent::MouseButtonPress。

许多事件类型需要的信息要比存储在普通 QEvent 对象中的信息多得多。例如,鼠标按键事件既需要保存是哪一个按键激发了该事件的信息,又需要保存发生该事件时鼠标指针所在的位置信息。这些额外信息就需要存储在专用的 QEvent 子类中,比如 QMouseEvent。

通过继承 QObject,事件通过它们的 event()函数来通知对象。在 QWidget 中的 event()实现把绝大多数常用类型的事件提前传递给特定的事件处理器,比如 mousePressEvent()、keyPressEvent()以及 paintEvent()。

在前几章中,当实现 MainWindow、IconEditor 和 Plotter 时,我们已经看到过许多事件处理器。在 QEvent 的参考文档中,还列出了很多其他类型的事件,并且也可以自己创建一些自定义事件类型和发布一些事件。在此将会回顾两种值得详述的常用事件类型:键盘事件和定时器事件。

通过重新实现 keyPressEvent()和 keyReleaseEvent(),就可以处理键盘事件了。Plotter 窗口部件就重新实现了 keyPressEvent()。通常,我们只需要重新实现 keyPressEvent()就可以了,因为用于表明释放重要性的键只是 Ctrl、Shift 和 Alt 这些修饰键,而这些键的状态可以在 keyPressEvent()中使用 QKeyEvent::modifiers()检测出来。例如,如果我们正打算实现一个 CodeEditor 窗口部件,那么要区分按下的键是 Home 还是 Ctrl + Home,它的 keyPressEvent()函数中的部分内容看起来应当是这样的:

```
void CodeEditor::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
    case Qt::Key_Home:
        if (event->modifiers() & Qt::ControlModifier) {
```

```
            goToBeginningOfDocument();
        } else {
            goToBeginningOfLine();
        }
        break;
    case Qt::Key_End:
        ...
    default:
        QWidget::keyPressEvent(event);
    }
}
```

Tab 键和 BackTab(Shift + Tab)键是两种特殊情况。在窗口部件调用 keyPressEvent()之前,QWidget::event()会先处理它们,它所包含的语义就是用于把焦点传递给焦点序列中的下一个或者上一个窗口部件。这种行为通常正是我们想要的,但是在 CodeEditor 窗口部件中,我们或许更愿意让 Tab 键起到缩进文本行的作用。于是,重新实现后的 event()看起来应当是下面的样子:

```
bool CodeEditor::event(QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
        if (keyEvent->key() == Qt::Key_Tab) {
            insertAtCurrentPosition('\t');
            return true;
        }
    }
    return QWidget::event(event);
}
```

如果该事件是一个按键事件,那么就把这个 QEvent 对象强制转换成 QKeyEvent 并且检查按下的是哪个键。如果按下的键是 Tab 键,就做一些处理并返回 true,告诉 Qt 已经把这个事件处理完毕了。如果返回的是 false,那么 Qt 将会把这个事件传递给它的父窗口部件来处理。

实现键绑定的一种更为高级的方法是使用 QAction。例如,如果在 CodeEditor 窗口部件中有 goToBeginningOfLine()和 goToBeginningOfDocument()两个公有槽,并且这个 CodeEditor 是用作 MainWindow 类的中央窗口部件的,那么本可以使用下面的代码来添加键绑定:

```
MainWindow::MainWindow()
{
    editor = new CodeEditor;
    setCentralWidget(editor);

    goToBeginningOfLineAction =
            new QAction(tr("Go to Beginning of Line"), this);
    goToBeginningOfLineAction->setShortcut(tr("Home"));
    connect(goToBeginningOfLineAction, SIGNAL(activated()),
            editor, SLOT(goToBeginningOfLine()));

    goToBeginningOfDocumentAction =
            new QAction(tr("Go to Beginning of Document"), this);
    goToBeginningOfDocumentAction->setShortcut(tr("Ctrl+Home"));
    connect(goToBeginningOfDocumentAction, SIGNAL(activated()),
            editor, SLOT(goToBeginningOfDocument()));
    ...
}
```

这样就可以很容易地把这个命令添加到菜单或者工具栏中了,就像第 3 章中看到的那样。如果这些命令没有出现在用户界面中,那么就可以用一个 QShortcut 对象替换这些 QAction 对象,这个类在 QAction 内部用来支持键绑定。

默认情况下,一旦把包含窗口部件的窗口激活,就可以在该窗口部件上使用 QAction 或 QShortcut 来启用所设置的绑定键。但使用 QAction::setShortcutContext()或者 QShortcut::setContext()可以改变这一点。

另外一种常用的事件类型是定时器事件。虽然绝大多数的其他事件类型的发生是因为用户的动作,但是定时器事件允许应用程序可以在一定的时间间隔后执行事件处理。定时器事件可以用来实现光标的闪烁和其他动画的播放,或者只简单地用作显示的刷新。

为了说明定时器事件,我们将会实现一个 Ticker 窗口部件,如图 7.1 所示。这个窗口部件显示了一串文本标语,它会每 30 毫秒向左移动一个像素。如果窗口部件比文本宽,那么文本将会被多次重复,直到能够填满整个窗口部件的宽度为止。

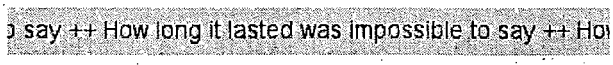


图 7.1 Ticker 窗口部件

以下给出的是头文件:

```
#ifndef TICKER_H
#define TICKER_H

#include <QWidget>

class Ticker : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)

public:
    Ticker(QWidget *parent = 0);

    void setText(const QString &newText);
    QString text() const { return myText; }
    QSize sizeHint() const;

protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);
    void showEvent(QShowEvent *event);
    void hideEvent(QHideEvent *event);

private:
    QString myText;
    int offset;
    int myTimerId;
};

#endif
```

我们重新实现了 Ticker 中的 4 个事件处理器,其中的 3 个以前不曾见到过:timerEvent()、showEvent()和 hideEvent()。

以下是它的实现文件:

```
#include <QtGui>

#include "ticker.h"

Ticker::Ticker(QWidget *parent)
    : QWidget(parent)
{
    offset = 0;
    myTimerId = 0;
}
```

构造函数把 offset 变量初始化为 0。用来绘制文本的 x 坐标值就取自于这个 offset 值。定时器的 ID 通常是非零的,所以可以使用 0 来表示定时器还没有启动。

```
void Ticker::setText(const QString &newText)
{
    myText = newText;
    update();
```

```
    updateGeometry();
}
```

setText()函数用来设置要显示的文本。它调用 update()强制执行一个重绘操作,并且调用 updateGeometry()通知对 Ticker 窗口部件负责的任意布局管理器,提示该窗口部件的大小发生了变化。

```
QSize Ticker::sizeHint() const
{
    return fontMetrics().size(0, text());
}
```

sizeHint()函数返回文本所需的空间大小,并以此作为窗口部件的理想尺寸。QWidget::fontMetrics()函数返回一个 QFontMetrics 对象,可以用这个对象查询并获得与这个窗口部件字体相关的信息。在这种情况下,可以询问给定文本所需的必要大小。[QFontMetrics::size()的第一个参数是一个标识符,对于那些比较简单的字符串来讲并不需要它,因而只给它传递了一个0值。]

```
void Ticker::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);

    int textWidth = fontMetrics().width(text());
    if (textWidth < 1)
        return;
    int x = -offset;
    while (x < width()) {
        painter.drawText(x, 0, textWidth, height(),
                         Qt::AlignLeft | Qt::AlignVCenter, text());
        x += textWidth;
    }
}
```

paintEvent()函数使用 QPainter::drawText()绘制文本。它使用 fontMetrics()确定文本在水平方向上所需要的空间,并且在考虑 offset 值的同时,多次绘制文本,直到能够填充整个窗口部件的宽度为止。

```
void Ticker::showEvent(QShowEvent * /* event */)
{
    myTimerId = startTimer(30);
}
```

showEvent()函数用来启动一个定时器。QObject::startTimer()调用会返回一个 ID 数字,可以在以后用这个数字识别该定时器。QObject 支持多个独立的定时器,每一个都可以有自己的时间间隔。在 startTimer()调用之后,大约每 30 毫秒 Qt 都会产生一个定时器事件。至于具体的时间精度,则取决于所在的操作系统。

我们本可以在 Ticker 的构造函数中完成 startTimer()的调用,但是只有在窗口部件实际可见的时候,才有必要保存由 Qt 产生的定时器事件的那些资源。

```
void Ticker::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        ++offset;
        if (offset >= fontMetrics().width(text()))
            offset = 0;
        scroll(-1, 0);
    } else {
        QWidget::timerEvent(event);
    }
}
```

系统每隔一定时间,都会调用一次 timerEvent()函数。它通过在 offset 上加 1 来模拟移动,从而形成文本宽度的连续滚动。然后,它使用 QWidget::scroll()把窗口部件的内容向左滚动一个像素。

本来也足可以调用 update()代替 scroll(),但使用 scroll()会更有效率,因为它只是简单地移动屏幕上已经存在的像素并且只对这个窗口部件的新显示区域(此时,只是一个 1 像素乘以宽度的像素条)产生一个绘制事件。

如果这个定时器事件不是我们所关注的那个定时器,就可以把它传递给基类。

```
void Ticker::hideEvent(QHideEvent * /* event */)
{
    killTimer(myTimerId);
    myTimerId = 0;
}
```

hideEvent()函数调用 QObject::killTimer()来停止该定时器。

定时器事件是一种低级事件,而且如果需要多个定时器时,保持对所有定时器 ID 的跟踪将会变得很麻烦。在这种情况下,通常更为简单的方式是为每一个定时器分别创建一个 QTimer 对象。QTimer 会在每个时间间隔发射 timeout()信号。QTimer 也提供了一个非常方便的接口,可用于单触发定时器(只触发一次的定时器),就像第 6 章(见 122 页)中看到的那样。

7.2 安装事件过滤器

Qt 事件模型一个非常强大的功能是:QObject 实例在看到它自己的事件之前,可以通过设置另外一个 QObject 实例先监视这些事件。

假定有一个由几个 QLineEdit 组成的 CustomerInfoDialog 窗口部件,并且我们想使用 Space(空格)键把光标移动到下一个 QLineEdit 中。这种非标准行为对于公司内部的某个应用程序来讲也许是非常合适的,因为使用它的用户可能都早已训练有素了。一种更为直接的解决方案是子类化 QLineEdit 并且重新实现 keyPressEvent(),由它调用 focusNextChild(),就像如下所示的形式:

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Space) {
        focusNextChild();
    } else {
        QLineEdit::keyPressEvent(event);
    }
}
```

这种方法有一个主要的缺点:如果在窗体中使用了好几种不同类型的窗口部件(例如,QComboBox 和 QSpinBox),我们也必须对它们逐一子类化,以便让它们能够实现相同的行为。一个更好的解决方案是让 CustomerInfoDialog 监视它的子窗口部件中键的按下事件并且在监视代码中实现所需的行为。这种方法可以通过使用事件过滤器来实现。创建一个事件过滤器包括如下两步过程:

1. 通过对目标对象调用 installEventFilter()来注册监视对象。
2. 在监视对象的 eventFilter()函数中处理目标对象的事件。

一个不错的注册监视对象的地方是在 CustomerInfoDialog 的构造函数中:

```
CustomerInfoDialog::CustomerInfoDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    firstNameEdit->installEventFilter(this);
    lastNameEdit->installEventFilter(this);
    cityEdit->installEventFilter(this);
    phoneNumberEdit->installEventFilter(this);
}
```

这个事件过滤器一旦注册,发送给 firstNameEdit、lastNameEdit、cityEdit 和 phoneNumberEdit 窗口部件的事件就会在它们到达目的地之前先被发送给 CustomerInfoDialog 的 eventFilter()函数。

以下是接收这些事件的 eventFilter()函数:

```
bool CustomerInfoDialog::eventFilter(QObject *target, QEvent *event)
{
    if (target == firstNameEdit || target == lastNameEdit
            || target == cityEdit || target == phoneNumberEdit) {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
            if (keyEvent->key() == Qt::Key_Space) {
                focusNextChild();
                return true;
            }
        }
    }
    return QDialog::eventFilter(target, event);
}
```

首先,需要检查一下目标窗口部件是不是 QLineEdit 中的一个。如果事件是一个按键事件,那么就把它强制转换为 QKeyEvent 并且检查按下的是哪个键。如果按下的键是空格键,那么就调用 focusNextChild()把焦点传递到焦点序列中的下一个窗口部件并且返回 true,告诉 Qt 已经处理了这个事件。如果返回 false,Qt 将会把这个事件发送给指定的目标对象,而在这个 QLineEdit 中产生一个虚假的空格插入操作。

如果窗口部件不是 QLineEdit,或者该事件不是一个空格按键事件,那么就把控制权传递给它的基类的 eventFilter()来执行。这个目标窗口部件也可能是某个基类(比如 QDialog)正在监控的窗口部件。(在 Qt 4.3 中,对于 QDialog 来说这不是什么问题。然而,其他的一些 Qt 窗口部件类,比如 QScrollArea,会因为各种各样的原因对它们自己的子窗口部件进行监控。)

Qt 提供了 5 个级别的事件处理和事件过滤方法。

1. 重新实现特殊的事件处理器

重新实现像 mousePressEvent()、keyPressEvent()和 paintEvent()这样的事件处理器是到现在为止最常用的事件处理方式。我们已经看到很多有关这种处理方式的例子了。

2. 重新实现 QObject::event()

通过 event()函数的重新实现,可以在这些事件到达特定的事件处理器之前处理它们。这种方式常用于覆盖 Tab 键的默认意义,就像在前面(见 130 页)看到的那样。这种方式也可以用于处理那些没有特定事件处理器的不常见类型的事件中(例如,QEvent::HoverEnter)。当重新实现 event()时,必须对那些没有明确处理的情况调用其基类的 event()函数。

3. 在 QObject 中安装事件过滤器

对象一旦使用 installEventFilter()注册过,用于目标对象的所有事件都会首先发送给这个监视对象的 eventFilter()函数。如果在同一个对象上安装了多个事件处理器,那么就会按照安装顺序逆序,从最近安装的到最先安装的,依次激活这些事件处理器。

4. 在 QApplication 对象中安装事件过滤器

一旦在 qApp(唯一的 QApplication 对象)中注册了事件过滤器,那么应用程序中每个对象的每个事件都会在发送到其他事件过滤器之前,先发送给这个 eventFilter()函数。这种处理方式对于调试是非常有用的。它也可以用来处理那些发送给失效窗口部件的鼠标事件,因为 QApplication 通常都会忽略这些事件。

5. 子类化 QApplication 并且重新实现 notify()

Qt 调用 QApplication::notify()来发送一个事件。重新实现这个函数是在事件过滤器得到所有事件之前获得它们的唯一方式。事件过滤器通常更有用,因为可以同时有多个事件过滤器,而 notify()函数却只能有一个。

很多事件类型,包括鼠标事件和按键事件,都可以对它们进行传递。如果在事件到达它的目标对象之前没有得到处理,或者也没有被它自己的目标对象处理,那么就会重复整个事件的处理过程,但这一次会把目标对象的父对象当作新的目标对象。这样一直继续下去,从父对象再到父对象的父对象,直到这个事件完全得到处理或者是到达了最顶层的对象为止。

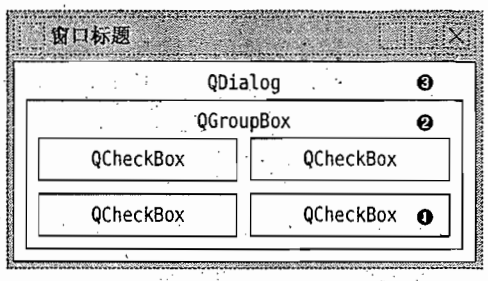


图 7.2 对话框中事件的传递

图 7.2 给出了一个按键事件是如何从对话框的子对象传递到父对象的过程。当用户按下一个键时,这个事件就会首先发送给拥有焦点的当前窗口部件,在本例中就是对话框右下角的 QCheckBox。如果 QCheckBox 没有处理这个事件,那么 Qt 就把它发送给 QGroupBox,直到最后的 QDialog 对象。

7.3 处理密集时的响应保持

当调用 QApplication::exec()时,就启动了 Qt 的事件循环。在开始的时候,Qt 会发出一些事件命令来显示和绘制窗口部件。在这之后,事件循环就开始运行,它不断检查是否有事件发生并且把这些事件发送给应用程序中的 QObject。

当处理一个事件时,也可能会同时产生一些其他的事件并且会将其追加到 Qt 的事件队列中。如果在处理一个特定事件上耗费的时间过多,那么用户界面就会变得无法响应。例如,在应用程序把一个文件保存到磁盘的过程中,直到文件保存完毕,才会处理那些由窗口系统产生的事件。在文件保存过程中,应用程序就不能响应来自窗口系统的重新绘制的请求。

在此情况下,一种解决方法就是使用多线程:一个线程用于处理应用程序的用户界面,另外一个线程则执行文件保存操作(或者任意的其他耗时操作)。这样的话,在保存文件的时候,应用程序的用户界面仍然可以保持响应。第 14 章中将看到如何实现这一方法。

一种更为简单的解决方法是在文件保存的代码中频繁调用 QApplication::processEvents()。这个函数告诉 Qt 处理所有那些还没有被处理的各类事件,然后再将控制权返还给调用者。实际上,QApplication::exec()就是一个不停调用 processEvents()函数的 while 循环。

以下是一个如何使用 processEvents()来保持用户界面响应的例子,该例基于 Spreadsheet(见 65 页)的文件保存功能的程序代码:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    QApplication::setOverrideCursor(Qt::WaitCursor);
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
```

```
            }
            qApp->processEvents();
        }
        QApplication::restoreOverrideCursor();
        return true;
    }
```

使用这种方法存在一个潜在问题,即用户也许在应用程序还在保存文件的时候就关闭了主窗口,或者甚至在保存文件的时候又一次单击了 File→Save,这样就可能会产生不可预料的后果。对于这个问题,最简单的解决方式是将

```
qApp->processEvents();
```

替换为

```
qApp->processEvents(QEventLoop::ExcludeUserInputEvents);
```

以告诉 Qt 忽略鼠标事件和键盘事件。

通常情况下,当需要发生一个长时间运行的操作时,我们希望能够显示一个 QProgressDialog。QProgressDialog 有一个进度条,它可以告诉用户应用程序中的这个操作目前的进度信息。QProgressDialog 还提供了一个 Cancel 按钮,它允许用户取消操作。以下是这种方法在保存Spreadsheet 中的一个文件时所使用的代码:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    QProgressDialog progress(this);
    progress.setLabelText(tr("Saving %1").arg(fileName));
    progress.setRange(0, RowCount);
    progress.setModal(true);

    for (int row = 0; row < RowCount; ++row) {
        progress.setValue(row);
        qApp->processEvents();
        if (progress.wasCanceled()) {
            file.remove();
            return false;
        }
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
    }
    return true;
}
```

我们以 NumRows 为总步数,创建一个 QProgressDialog。然后,对于每一行,都调用 setValue()来更新这个进度条。QProgressDialog 可以自动地计算出进度的百分比,该值等于当前进度值除以总步数。我们调用 QApplication::processEvents()来处理任意的重绘事件或者用户任意的鼠标单击或者按键事件(例如,要允许用户可以单击 Cancel)。如果用户单击了 Cancel,就应该放弃保存这个文件并且将其删除。

我们没有对 QProgressDialog 调用 show(),这是因为进度对话框会自动调用它。如果这个操作可以很快完成(比如,可能是由于文件很小,或者可能是计算机的速度特别快),QProgressDialog 就会检测到这些情况并且不再显示出来。

除使用多线程和 QProgressDialog 之外,还有一种处理长时间运行操作的完全不同的方法:不是在用户请求的时候执行处理,而是一直推迟到应用程序空闲下来的时候才处理。如果该处理可以

被安全中断后继续,那么就可以使用这种方法了,因为我们并不能事先知道应用程序要多长时间后才能闲置下来。

在 Qt 中,通过使用一个 0 毫秒定时器就可以实现这种方法。只要没有其他尚待处理的事件,就可以触发这个定时器。以下是一个 timerEvent()实现的例子,其中给出了程序空闲时的处理方法:

```
void Spreadsheet::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        while (step < MaxStep && !qApp->hasPendingEvents()) {
            performStep(step);
            ++step;
        }
    } else {
        QTableWidget::timerEvent(event);
    }
}
```

如果 hasPendingEvents()的返回值是 true,就停止处理并且把控制权交还给 Qt。当 Qt 处理完所有事件后,就会重新恢复这项操作。

第8章 二维图形

Qt的二维图形引擎是基于QPainter类的。QPainter既可以绘制几何形状(点、线、矩形、椭圆、弧形、弦形、饼状图、多边形和贝塞尔曲线),也可以绘制像素映射、图像和文字。此外,QPainter还支持一些高级特性,例如反走样(针对文字和图形边缘)、像素混合、渐变填充和矢量路径等。QPainter也支持线性变换,例如平移、旋转、错切和缩放。

QPainter可以画在"绘图设备"上,例如QWidget、QPixmap、QImage或者QSvgGenerator。QPainter也可以与QPrinter一起使用来打印文件和创建PDF文档。这意味着通常可以用相同的代码在屏幕上显示数据,也可以生成打印形式的报告。

重新实现QWidget::paintEvent()可用于定制窗口部件,并且随心所欲地控制它们的外观,就像第5章中看到的那样。定制预定义Qt窗口部件的外观,可以指定某一风格的表单或者创建一个QStyle的子类,第19章将介绍这两种方法。

一项普通的需求是在二维画板上显示大量的、轻量级的、可与用户交互的、任意形状的项。Qt 4.2围绕着QGraphicsView、QGraphicsScene和QGraphicsItem类引入了一个全新的"图形视图"体系。这个体系提供了一个操作基于项的图形的高级接口,并且支持用户对项的操作,包括移动、选取、分组。项本身依然用QPainter画出,并且可以独立变换。本章稍后将描述这一体系。

可以使用OpenGL命令来代替QPainter。OpenGL是一个绘制三维图形的标准库。第20章将介绍如何使用QtOpenGL模块,该模块简化了OpenGL代码与Qt应用程序之间的集成。

8.1 用QPainter绘图

要想在绘图设备(一般是窗口部件)上绘图,只需创建一个QPainter,再将指针传到该设备中。例如:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    ...
}
```

使用QPainter的draw...()函数,可以绘制各种各样的形状。图8.1列出了其中最重要的一些函数。绘制的效果取决于QPainter的设置。一些值是从设备中取得的,然而有些被初始化成默认值。三个主要的设置是画笔、画刷和字体:

- 画笔用来画线和边缘。它包含颜色、宽度、线型、拐点风格以及连接风格。画笔的风格分别如图8.2和图8.3所示。
- 画刷用来填充几何形状的图案。它一般由颜色和风格组成,但同时也可以是纹理(一个不断重复的图像)或者是一个渐变。画刷风格如图8.4所示。
- 字体用来绘制文字。字体有很多属性,包括字体族和磅值大小。

可以随时调用QPen、QBrush或者QFont对象的setPen()、setBrush()和setFont()来修改这些设置。

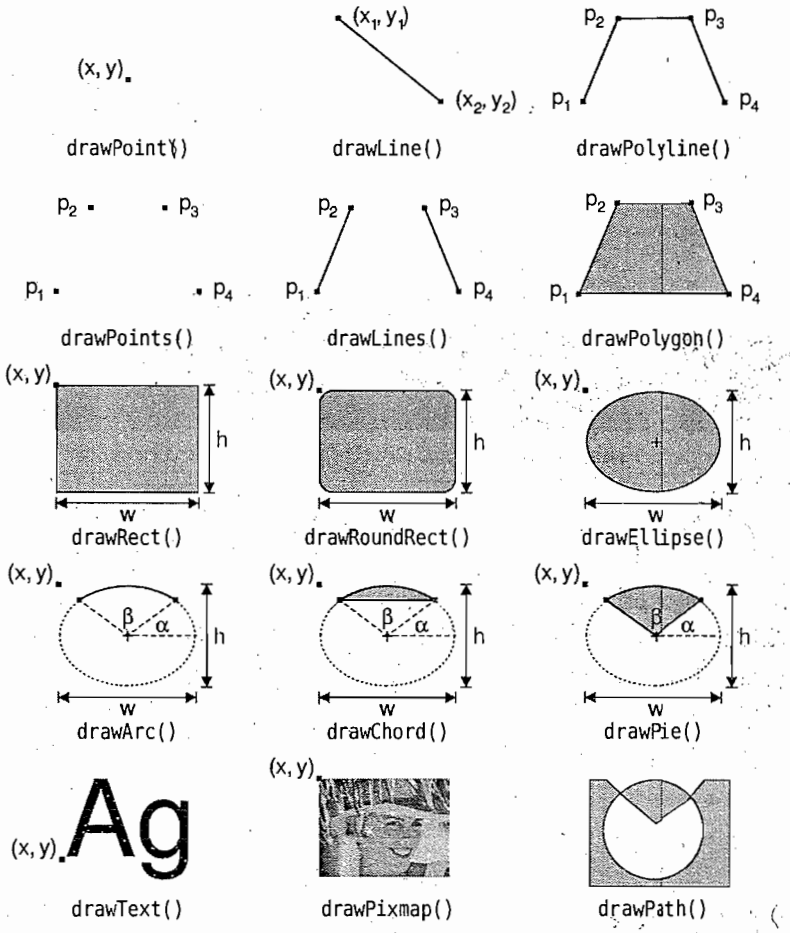


图 8.1　QPainter 最常用的各种 draw...()函数

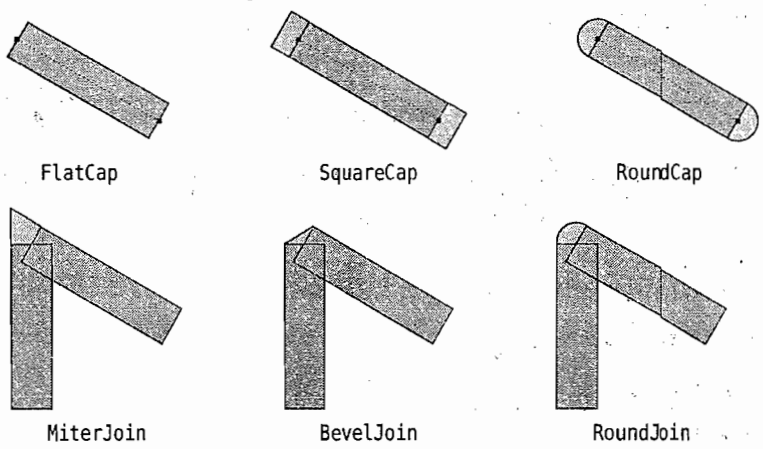


图 8.2　各种拐点风格和连接风格

来看几个实际的例子。下面是绘制图 8.5(a)中的椭圆的代码：

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine, Qt::RoundCap));
painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
painter.drawEllipse(80, 80, 400, 240);
```

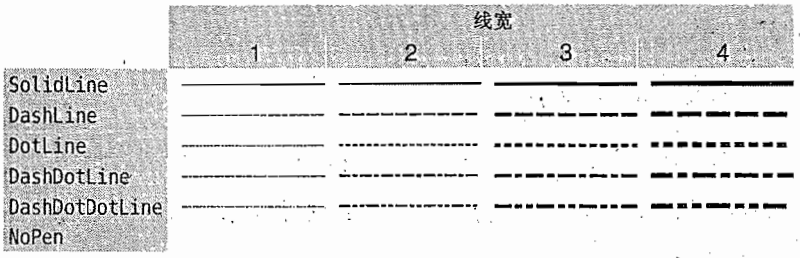


图 8.3 各种线型风格

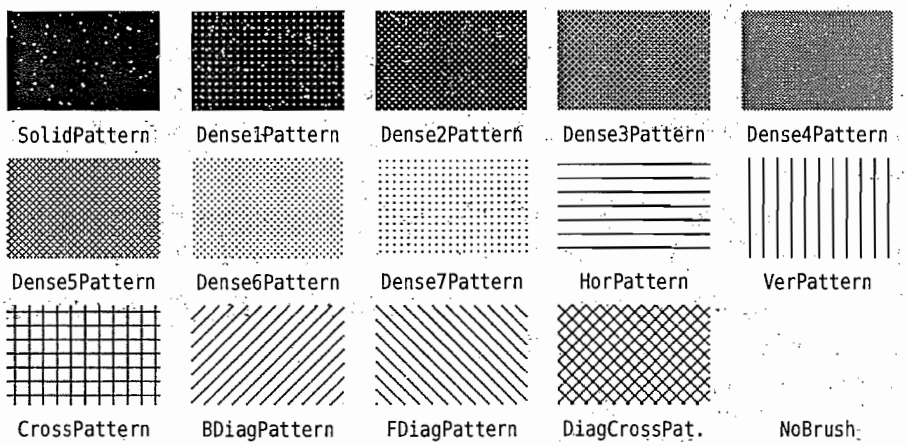
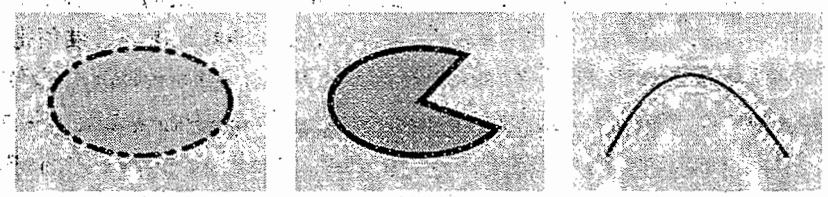


图 8.4 各种预定义的画刷风格



(a) 椭圆 (b) 饼状图 (c) 贝塞尔曲线

图 8.5 各种几何图形示例

调用 setRenderHint()可以启用反走样,它会告诉 QPainter 用不同的颜色强度绘制边框以减少视觉扭曲,这种扭曲一般会在边框转换为像素的时候发生。由此生成的结果是可以在支持这一特性的平台和设备上得到平滑的边缘。

下面是绘制图 8.5(b)的饼状图所用的代码:

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 15, Qt::SolidLine, Qt::RoundCap,
                    Qt::MiterJoin));
painter.setBrush(QBrush(Qt::blue, Qt::DiagCrossPattern));
painter.drawPie(80, 80, 400, 240, 60 * 16, 270 * 16);
```

drawPie()的最后两个参数以 1/16 度为单位。

下面是绘制图 8.5(c)的三次贝塞尔曲线时所使用的代码:

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);

QPainterPath path;
```

```
path.moveTo(80, 320);
path.cubicTo(200, 80, 320, 80, 480, 320);

painter.setPen(QPen(Qt::black, 8));
painter.drawPath(path);
```

QPainterPath 类可以通过连接基本的图形元素来确定任意的矢量形状:直线、椭圆、多边形、弧形、贝塞尔曲线和其他的绘制路径。绘制路径是基本的图元,从这个意义上来说,任何图形或图形组合都可以用绘制路径描述。

路径可以确定一个边缘,由边缘锁定的区域可以用画刷来填充。在图 8.5(c)的例子中没有设置画刷,所以只能看见边缘。

这三个例子利用内置的画刷模式(Qt::SolidPattern、Qt::DiagCrossPattern 和 Qt::NoBrush)。在现代应用中,渐变填充已成为单色填充的流行替代品。渐变填充利用颜色插值使得两个或更多颜色之间能够平滑过渡。它们常被用来创建三维效果,Plastique 和 Cleanlooks 风格就是使用渐变来渲染 QPushButton 的。

Qt 支持三种类型的渐变:线性渐变、锥形渐变和辐射渐变。下一节中的烤箱定时器的例子就是在一个独立的窗口部件上结合了这三种类型的渐变,从而使其看起来更加逼真。

- 线性渐变(linear gradient)由两个控制点定义,连接这两点的线上有一系列的颜色断点。例如,图 8.6 的线性渐变由以下的代码创建:

```
QLinearGradient gradient(50, 100, 300, 350);
gradient.setColorAt(0.0, Qt::white);
gradient.setColorAt(0.2, Qt::green);
gradient.setColorAt(1.0, Qt::black);
```

 在两个控制点之间的不同位置指定三种颜色。位置用 0 和 1 之间的浮点数来指定,0 对应第一个控制点,1 对应第二个控制点。两个指定断点之间的颜色由线性插值得出。
- 辐射渐变(radial gradient)由一个中心点(x_c, y_c)、半径 r、一个焦点(x_f, y_f),以及颜色断点定义。中心点和半径定义一个圆。颜色从焦点向外扩散,焦点可以是中心点或者圆内的其他点。
- 锥形渐变(conical gradient)由一个中心点(x_c, y_c)和一个角度 a 定义。颜色在中心点周围像钟表的秒针掠过一样扩散。

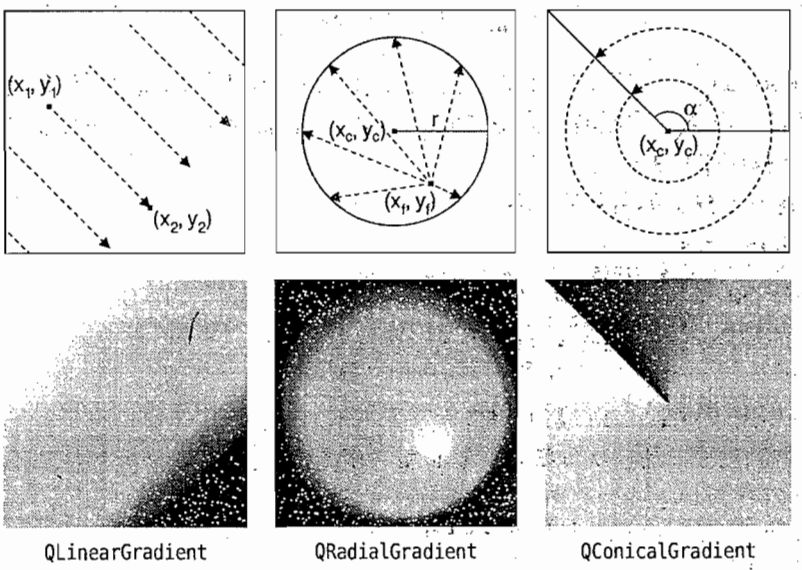


图 8.6 QPainter 的各种渐变画刷

目前,我们提到了 QPainter 的画笔、画刷和字体设置。另外,QPainter 还有其他影响图形和文字绘制方式的设置:

- 当背景模式是 Qt::OpaqueMode(默认设置是 Qt::TransparentMode)时,背景画刷可以用来填充几何图形的背景(画刷模式下)、文字或者位图。
- 画刷的原点是画刷模式的启动点,通常是窗口部件的左上角。
- 剪切区域是设备的绘图区域。在剪切区域以外绘图不起作用。
- 视图、窗口和世界矩阵决定了如何将 QPainter 的逻辑坐标映射到设备的物理绘图坐标。默认情况下,为了使逻辑坐标和物理坐标保持一致,这些是设置好的。下一节将描述坐标系统。
- 复合模式指定新绘制的像素如何跟绘图设备上已经显示的像素相互作用。默认方式是"source over",这种情况下像素会被 alpha 混合在存在的像素上。只有特定的设备支持这种模式,本章稍后将描述这种模式。

可以通过调用 save()而随时将设备的当前状态存入一个内部堆栈,稍后通过调用 restore()恢复。这对我们如果想临时修改一些设备的设置然后恢复到以前的状态会很有用,会在下一节见到这种情况。

8.2 坐标系统变换

在 QPainter 的默认坐标系中,点(0,0)位于绘图设备的左上角,x 坐标向右增长,y 坐标向下增长。默认坐标系的每个像素占 1×1 大小的区域。

理论上,像素的中心取决于半像素坐标。例如,窗口部件的左上角像素覆盖了点(0,0)到点(1,1)的区域,它的中心在(0.5,0.5)位置。如果告诉 QPainter 绘制一个像素,例如(100,100),它会相应地在两个方向做 +0.5 的偏移,使得像素点的中心位置在(100.5,100.5)。

这一差别初看起来理论性很强,但它在实践中却很重要。首先,只有当反走样无效时(默认情况)才偏移 +0.5;如果反走样有效,并且我们试图在(100,100)的位置绘制一个黑色的像素,实际上 QPainter 会为(99.5,99.5)、(99.5,100.5)、(100.5,99.5)和(100.5,100.5)四个像素点着浅灰色,给人的印象是一个像素正好位于四个像素的重合处。如果不需要这种效果,可以通过指定半像素坐标或者通过偏移 QPainter(+0.5, +0.5)来避免这种效果的出现。

当绘制图形时,例如线、矩形和椭圆,可以使用相似的规则。图 8.7 表明当反走样关闭时调用 drawRect(2,2,6,5)时不同画笔宽度产生的变化。值得特别注意的是,6×5 宽度为 1 的矩形可以有效地覆盖 7×6 的区域。这不同于旧的工具,包括早期的 Qt 版本,其本质是要支持真正可缩放的、与分辨率无关的矢量图。图 8.8 表明当反走样打开时调用 drawRect(2,2,6,5)产生的结果,图 8.9 表明当指定半像素坐标时产生的结果。

既然已经了解了默认的坐标系统,就可以进一步看一下怎样使用 QPainter 的视口、窗口、世界矩阵。(在上下文中,术语"窗口"说的不是顶层窗口部件窗口,"视口"跟 QScrollArea 的视口也没有关系。)

视口和窗口密不可分。视口是物理坐标系下指定的任意矩形。窗口也是指同一矩形,只不过是在逻辑坐标系下。当绘制图形时,在逻辑坐标系下指定的点,这些坐标都是基于当前的窗口-视口设定并以线性代数的方式转换为物理坐标的。

默认情况下,视口和窗口都被设置成设备的矩形。例如,如果设备是 320×200 的矩形,视口和窗口都是左上角为(0,0)的 320×200 的相同矩形。这种情况下,逻辑坐标系和物理坐标系是一致的。

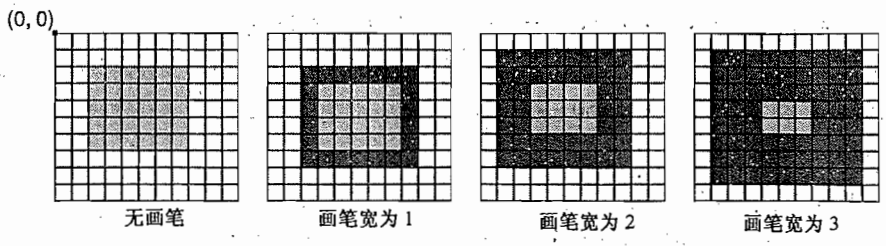


图 8.7　未经反走样处理的 drawRect(2,2,6,5)函数的绘制效果

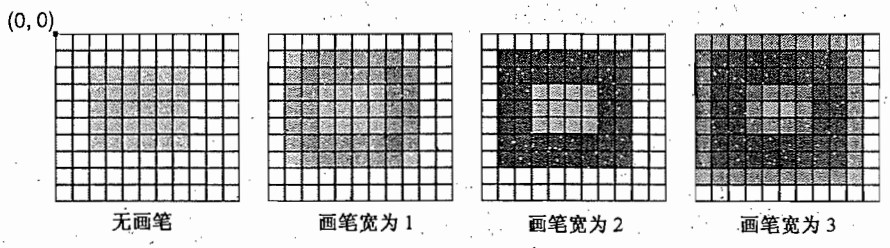


图 8.8　反走样处理的 drawRect(2,2,6,5)函数的绘制效果

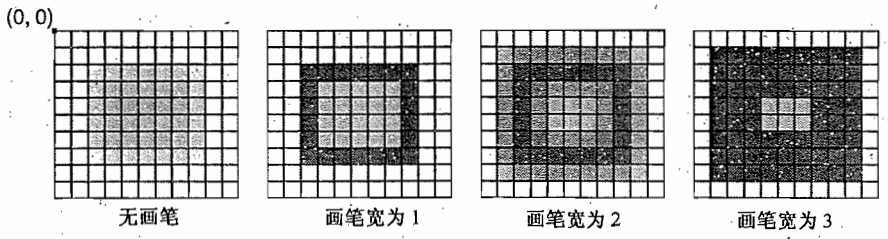


图 8.9　反走样处理的 drawRect(2.5,2.5,6,5)函数的绘制效果

这种窗口-视口机制对于编写独立于绘制设备大小和分辨率的绘制代码是很有用的。例如，如果想让逻辑坐标从(-50,-50)到(+50,+50)，并且(0,0)在中间，可以这样设置窗口：

```
painter.setWindow(-50, -50, 100, 100);
```

(-50,-50)指定了原点，(100,100)指定了宽和高。这意味着逻辑坐标(-50,-50)对应物理坐标(0,0)，而逻辑坐标(+50,+50)对应物理坐标(320,200)，如图 8.10 所示。这个例子没有改变视口。

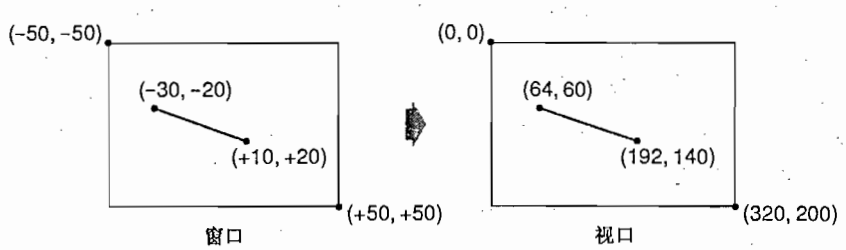


图 8.10　逻辑坐标向物理坐标的转换

现在来介绍世界变换。世界变换(world transform)是在窗口-视口转换之外使用的变换矩阵。它允许移动、缩放、旋转或者拉伸绘制的项。例如，如果想以 45°角绘制文本，可以使用这段代码：

```
QTransform transform;
transform.rotate(+45.0);
painter.setWorldTransform(transform);
painter.drawText(pos, tr("Sales"));
```

传递给 drawText()的逻辑坐标会被世界变换转换,然后使用窗口-视口设置映射到物理。

如果进行了多次变换,它们会按照给定的顺序生效。例如,如果想要使用点(50,50)作为旋转的中心点,可以移动窗口到(+50,+50),执行旋转,然后把窗口移回到原来的初始位置。

```
QTransform transform;
transform.translate(+50.0, +50.0);
transform.rotate(+45.0);
transform.translate(-50.0, -50.0);
painter.setWorldTransform(transform);
painter.drawText(pos, tr("Sales"));
```

坐标变换的一种更为简单的方式是使用 QPainter 的 translate()、scale()、rotate()和 shear()这些简便函数。

```
painter.translate(-50.0, -50.0);
painter.rotate(+45.0);
painter.translate(+50.0, +50.0);
painter.drawText(pos, tr("Sales"));
```

如果想重复使用相同的变换,可以把它们保存到一个 QTransform 对象中,这样会更高效。在需要变换的时候,再把世界变换设置到绘制器上。

为了描述绘制器的变换,我们将分析如图 8.11 和图 8.12 所示 OvenTimer 窗口部件的代码。这个 OvenTimer 模仿烤箱的定时器,它是烤箱内置的钟表。用户可以单击刻度来设置持续时间。转轮会自动地逆时针转到 0,OvenTimer 在这一点发射 timeout()信号。

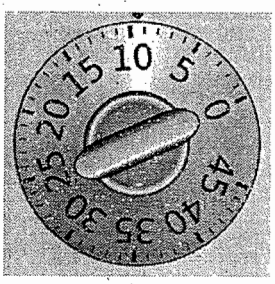


图 8.11 OvenTimer 窗口部件

```
class OvenTimer : public QWidget
{
    Q_OBJECT

public:
    OvenTimer(QWidget *parent = 0);

    void setDuration(int secs);
    int duration() const;
    void draw(QPainter *painter);

signals:
    void timeout();

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);

private:
    QDateTime finishTime;
    QTimer *updateTimer;
    QTimer *finishTimer;
};
```

OvenTimer 类继承自 QWidget,重新实现了两个虚函数:paintEvent()和 mousePressEvent()。

```
const double DegreesPerMinute = 7.0;
const double DegreesPerSecond = DegreesPerMinute / 60;
const int MaxMinutes = 45;
const int MaxSeconds = MaxMinutes * 60;
const int UpdateInterval = 5;
```

在 oventimer.cpp 中,定义了几个控制烤箱定时器外观的常量。

```
OvenTimer::OvenTimer(QWidget *parent)
    : QWidget(parent)
{
    finishTime = QDateTime::currentDateTime();
    updateTimer = new QTimer(this);
    connect(updateTimer, SIGNAL(timeout()), this, SLOT(update()));

    finishTimer = new QTimer(this);
    finishTimer->setSingleShot(true);
    connect(finishTimer, SIGNAL(timeout()), this, SIGNAL(timeout()));
    connect(finishTimer, SIGNAL(timeout()), updateTimer, SLOT(stop()));

    QFont font;
    font.setPointSize(8);
    setFont(font);
}
```

在构造函数中,创建了两个 QTimer 对象:updateTimer 用来每隔 5 秒刷新窗口部件的外观,finishTimer 在定时器达到 0 时发射 timeout()信号。finishTimer 只需要执行一次,因此我们调用 setSingleShot(true)。默认情况下,定时器会重复触发直到它们被停止或销毁。最后一个 connect()调用是一个优化,目的是当定时器停止运行时停止更新窗口部件。

在构造函数的末尾,我们把用来绘制窗口部件的字体的磅值设置为 8 磅。这保证了显示的数字大小大体一致。

```
void OvenTimer::setDuration(int secs)
{
    secs = qBound(0, secs, MaxSeconds);

    finishTime = QDateTime::currentDateTime().addSecs(secs);

    if (secs > 0) {
        updateTimer->start(UpdateInterval * 1000);
        finishTimer->start(secs * 1000);
    } else {
        updateTimer->stop();
        finishTimer->stop();
    }
    update();
}
```

setDuration()函数设置了烤箱定时器的持续时间为给定的秒数。使用 Qt 的 qBound()函数意味着可以不必写这样的代码:

```
if (secs < 0) {
    secs = 0;
} else if (secs > MaxSeconds) {
    secs = MaxSeconds;
}
```

我们通过在当前时间[从 QDataTime::currentDateTime()获得]加上间隔时间得到结束时间,并且把它保存到 finishTime 私有变量中。在结束的地方,我们调用 update(),用新的间隔时间重绘窗口部件。

finishTime 变量的类型是 QDateTime。这个变量保存日期和时间,避免了由于当前时间是在午夜之前并且完成时间是在午夜之后而产生的折回缺陷(wrap-around bug)。

```
int OvenTimer::duration() const
{
    int secs = QDateTime::currentDateTime().secsTo(finishTime);
    if (secs < 0)
        secs = 0;
    return secs;
}
```

duration()返回定时器完成前剩余的秒数。如果定时器未激活,返回 0。

```
void QvenTimer::mousePressEvent(QMouseEvent *event)
{
    QPointF point = event->pos() - rect().center();
    double theta = std::atan2(-point.x(), -point.y()) * 180.0 / M_PI;
    setDuration(duration() + int(theta / DegreesPerSecond));
    update();
}
```

当用户点击窗口部件时,我们使用一个巧妙且有效的数学公式找到最近的刻度,并且使用这个结果来设置新的持续时间,然后安排一个重绘。用户点过的刻度移到了顶部,然后随着时间逆时针移动到 0。

```
void QvenTimer::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);

    int side = qMin(width(), height());

    painter.setViewport((width() - side) / 2, (height() - side) / 2,
                        side, side);
    painter.setWindow(-50, -50, 100, 100);

    draw(&painter);
}
```

在 paintEvent()函数中,我们把视口设置成窗口部件中最大的正方形,把窗口设置成(-50,-50,100,100)的矩形,也就是说,100×100 的矩形从(-50,-50)扩展到(+50,+50)。qMin()模板函数返回两个参数中最小的一个。然后调用 draw()函数实际执行绘图。

如果没有设置视口为正方形,当窗口部件被缩放为非正方形的矩形时,烤箱定时器就会变为椭圆。为了避免这种变形,必须把视口和窗口设置成具有相同纵横比的矩形,这种效果见图 8.12。

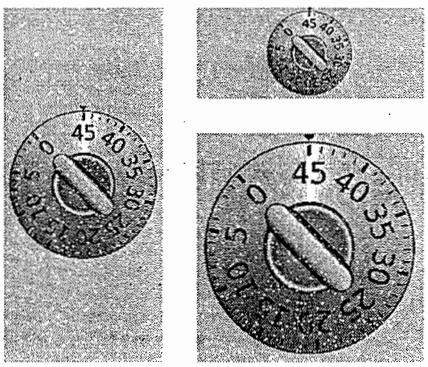


图 8.12 三种不同尺寸的 QvenTimer 窗口部件

现在,来看看绘图的代码:

```
void QvenTimer::draw(QPainter *painter)
{
    static const int triangle[3][2] = {
        { -2, -49 }, { +2, -49 }, { 0, -47 }
    };
    QPen thickPen(palette().foreground(), 1.5);
    QPen thinPen(palette().foreground(), 0.5);
    QColor niceBlue(150, 150, 200);

    painter->setPen(thinPen);
    painter->setBrush(palette().foreground());
    painter->drawPolygon(QPolygon(3, &triangle[0][0]));
```

首先,在窗口部件顶部的0位置绘制一个小三角形。这个三角形的三个坐标都是由代码直接给定的,我们用 drawPolygon()绘制它。

窗口-视口的方便之处就在于可以在绘制命令中使用硬编码,却能得到很好的缩放效果。

```
QConicalGradient coneGradient(0, 0, -90.0);
coneGradient.setColorAt(0.0, Qt::darkGray);
coneGradient.setColorAt(0.2, niceBlue);
coneGradient.setColorAt(0.5, Qt::white);
coneGradient.setColorAt(1.0, Qt::darkGray);

painter->setBrush(coneGradient);
painter->drawEllipse(-46, -46, 92, 92);
```

我们绘制外面的圆,用锥形渐变填充。锥形的中心点位于(0,0),角度是-90°。

```
QRadialGradient haloGradient(0, 0, 20, 0, 0);
haloGradient.setColorAt(0.0, Qt::lightGray);
haloGradient.setColorAt(0.8, Qt::darkGray);
haloGradient.setColorAt(0.9, Qt::white);
haloGradient.setColorAt(1.0, Qt::black);

painter->setPen(Qt::NoPen);
painter->setBrush(haloGradient);
painter->drawEllipse(-20, -20, 40, 40);
```

我们用辐射渐变填充内部的圆。渐变的中心点和焦点都在(0,0)。渐变的半径是20。

```
    QLinearGradient knobGradient(-7, -25, 7, -25);
    knobGradient.setColorAt(0.0, Qt::black);
    knobGradient.setColorAt(0.2, niceBlue);
    knobGradient.setColorAt(0.3, Qt::lightGray);
    knobGradient.setColorAt(0.8, Qt::white);
    knobGradient.setColorAt(1.0, Qt::black);

    painter->rotate(duration() * DegreesPerSecond);
    painter->setBrush(knobGradient);
    painter->setPen(thinPen);
    painter->drawRoundRect(-7, -25, 14, 50, 99, 49);

    for (int i = 0; i <= MaxMinutes; ++i) {
        if (i % 5 == 0) {
            painter->setPen(thickPen);
            painter->drawLine(0, -41, 0, -44);
            painter->drawText(-15, -41, 30, 30,
                              Qt::AlignHCenter | Qt::AlignTop,
                              QString::number(i));
        } else {
            painter->setPen(thinPen);
            painter->drawLine(0, -42, 0, -44);
        }
        painter->rotate(-DegreesPerMinute);
    }
}
```

我们调用 rotate()来旋转绘图器的坐标系统。在旧的坐标系统中,0-minute 的标记在最上面;现在,这个标记移到了对应着剩余时间的位置。在这个旋转之后,我们绘制了矩形把手,因为它的方位依赖于旋转的角度。

在 for 循环中,我们在外面圆的边缘绘制了标记,并且绘制的数字都是5分钟的倍数。文本被放置在标记下边一个不可见的矩形里。在每个迭代的最后,顺时针旋转绘图器7°,这相当于1分钟。下次将把标记绘制在圆的其他位置,尽管传给 drawLine()和 drawText()的坐标总是一样的。

for 循环中的代码有一个小缺陷,如果执行了更多的迭代,这一问题会变得很明显。每次调用 rotate(),就高效地用一个旋转矩阵去乘当前的世界变换,从而创建一个新的世界变换。浮点数的

舍入误差不断累积,得到了越来越不准确的世界变换。用下面的办法重写这段代码可以避免这个问题,使用 save()和 restore()函数为每次迭代保存和加载原始的矩阵:

```
for (int i = 0; i <= MaxMinutes; ++i) {
    painter->save();
    painter->rotate(-i * DegreesPerMinute);

    if (i % 5 == 0) {
        painter->setPen(thickPen);
        painter->drawLine(0, -41, 0, -44);
        painter->drawText(-15, -41, 30, 30,
                          Qt::AlignHCenter | Qt::AlignTop,
                          QString::number(i));
    } else {
        painter->setPen(thinPen);
        painter->drawLine(0, -42, 0, -44);
    }
    painter->restore();
}
```

实现烤箱定时器的另外一个办法是自己计算(x,y)位置,使用 sin()和 cos()函数找到圆上的位置。但之后将仍然需要利用移动和旋转并以一定的角度来绘制文本。

8.3 用 QImage 高质量绘图

绘图时,我们可能需要面对速度和准确率的折中问题。例如,在 X11 和 Mac OS X 系统中,要在 QWidget 或 QPixmap 上绘图,需要依赖于平台自带的绘图引擎。在 X11 上,这保证了与 X 服务器的通信限制在一个最小集:仅仅发送绘图命令,而不是图像数据。这一方法的主要缺点是 Qt 受限于平台的内在支持:

- 在 X11 上,类似反走样以及对分数坐标的支持只有当 X 服务器上存在 X 渲染扩展时才有效。
- 在 Mac OS X 上,内置的走样绘图引擎使用与 X11 和 Windows 不同的算法绘制多边形,绘制结果也稍有不同。

当准确率比效率更为重要时,我们可以画到一个 QImage 上,然后把结果复制到屏幕上。这样可以总是使用 Qt 自己内置的绘图引擎,在所有平台上得到同样的结果。唯一的限制就是 QImage 在被创建时会用到 QImage::Format_RGB32 或者 QImage::Format_ARGB32_Premultiplied 参数。

预乘 ARGB32 格式与常规的 ARGB32 格式(0xAARRGGBB)差不多是一样的,不同之处在于红、绿和蓝通道自左乘 alpha(透明)通道。这意味着,RGB 通道的值,一般是从 0x00 到 0xFF,变换为从 0x00 到透明通道的值。例如,50% 透明的蓝色用 ARGB32 格式表示为 0x7F0000FF,但用预乘 ARGB32 格式表示为 0x7F00007F。同样,ARGB32 格式的 75% 透明的黑绿表示为 0x3F008000,用预乘 ARGB32 格式表示为 0x3F002000。

假设我们想应用反走样绘制一个窗口部件,并且想在没有 X 渲染扩展的 X11 系统上获得很好的结果。原始的 paintEvent()处理器,反走样依赖于 X 渲染的代码类似于:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    draw(&painter);
}
```

下面的代码给出了如何使用 Qt 的平台无关的绘图引擎重写窗口部件的 paintEvent()函数:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QImage image(size(), QImage::Format_ARGB32_Premultiplied);
    QPainter imagePainter(&image);
    imagePainter.initFrom(this);
    imagePainter.setRenderHint(QPainter::Antialiasing, true);
    imagePainter.eraseRect(rect());
    draw(&imagePainter);
    imagePainter.end();

    QPainter widgetPainter(this);
    widgetPainter.drawImage(0, 0, image);
}
```

我们以预乘 ARGB32 模式创建一个跟窗口部件大小一致的 QImage，以及一个 QPainter 引用此图像。调用 initForm()函数初始化画笔、背景和字体。我们照常使用 QPainter 绘图，在结尾处，仍然使用 QPainter 对象把图像复制到窗口部件上。这种方法在所有平台上产生同样高质量的结果，对于字体渲染，依赖于安装的字体库。

Qt 图形引擎的一个特别强大的特性是它支持复合模式。这规范了源和目的像素如何在绘制时复合在一起。该模式可用于所有的绘制操作，包括画笔、画刷、渐变以及图像绘制。

默认的复合模式是 QImage::CompositionMode_SourceOver，这意味着源像素（正在绘制的像素）被混合在目的像素（已存在的像素）上，这样，源图像的透明部分给我们以透明效果。图 8.13 列举了应用不同模式在（玻璃钢化时的）风嘴印图像（目的图像）上绘制一个半透明的蝴蝶（源图像）的效果。

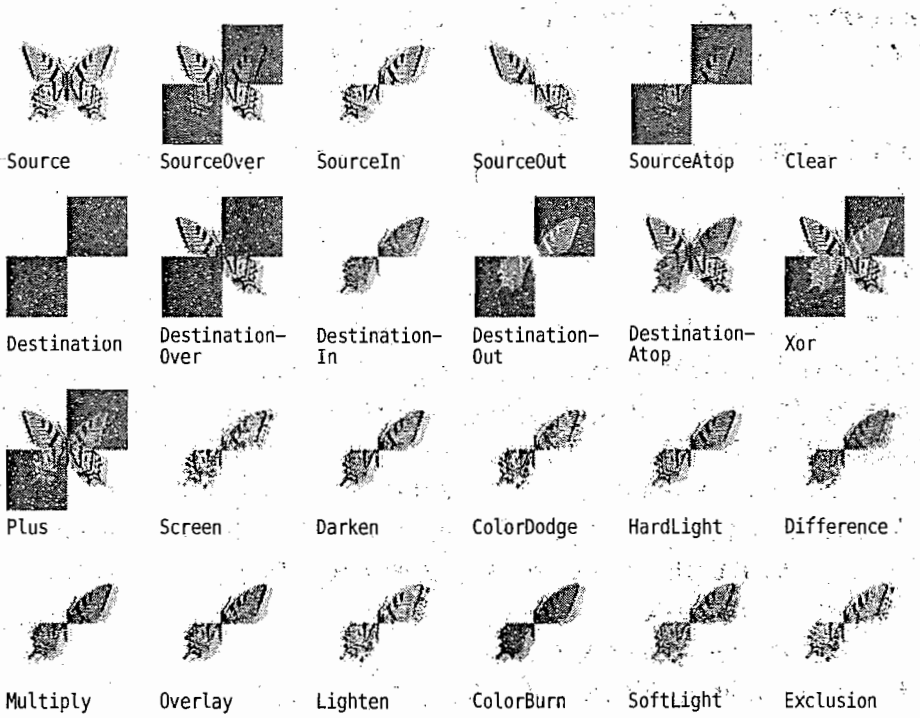


图 8.13　QPainter 的各种复合模式

使用 QPainter::setCompositionMode()可以设置各种复合模式。例如，下面就是如何设置蝴蝶和风嘴印图像 XOR 复合模式的代码：

```
QImage resultImage = checkerPatternImage;
QPainter painter(&resultImage);
painter.setCompositionMode(QPainter::CompositionMode_Xor);
painter.drawImage(0, 0, butterflyImage);
```

值得注意的是,QImage::CompositionMode_Xor操作也会影响到透明通道。这意味着,如果白色(0xFFFFFFFF)对自己做 XOR 复合,会得到透明色(0x00000000),而不是黑色(0xFF000000)。

8.4 基于项的图形视图

对于用户自定义窗口部件和绘制一个或者几个项来说,使用 QPainter 是理想的。在绘图中,如果需要处理从几个到几万的项时,而且要求用户能够单击、拖动和选取项,Qt 的视图类提供了对这一问题的解决方案。

Qt 的视图体系包括一个由 QGraphicsScene 充当的场景和一些由 QGraphicsItem 的子类充当的场景中的项。场景(以及它的项)在视图中显示,这样用户就可以看到了,它由 QGraphicsView 类充当。同一场景可以在多个视图中显示——例如,便于部分地显示一个大的场景,或者以不同的变换来显示场景。参见示意图 8.14。

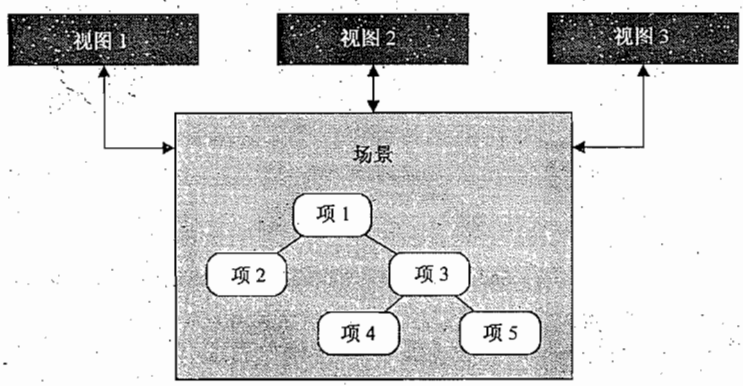


图 8.14 一个场景可用于多个视图中

图 8.15 提供了几个预定义的 QGraphicsItem 子类,包括 QGraphicsLineItem、QGraphicsPixmapItem、QGraphicsSimpleTextItem(应用于纯文本),以及 QGraphicsTextItem(应用于多文本)。也可以创建自己的 QGraphicsItem 子类,稍后将在本节中看到①。

QGraphicsScene 是一个图形项的集合。一个场景有三层:背景层(background layer)、项层(item layer)和前景层(foreground layer)。背景层和前景层通常由 QBrush 指定,但也可能需要重新实现 drawBackground()和 drawForeground(),以便可以实现完全控制。如果想用一个图片作为背景,可以简单地创建该图片作为 QBrush 纹理。前景画刷可以设置成半透明的白色,给人一种褪色的效果,或者设置成交叉模式,提供一种格子覆盖的效果。

场景可以告诉我们哪些项是重叠的,哪些是被选取的,以及哪些是在一个特定的点处,或者在一个特定的区域内。场景中的项或者是最高层的项(场景就是其父对象),或者是子项(它们的父对象是另外的项)。任何应用于项的变换都会自动地应用于子对象。

视图体系提供了两种分组项的方法。一种方法是简单地使一个项成为另一个项的子项。另

① Qt 4.4 有望支持在图形视图中添加窗口部件,如同视图项一样,包括对它们进行坐标变换的能力。

外一种方法是使用 QGraphicsItemGroup。把一个项添加到组中不会引起任何变换,这些组可以很方便地处理大量的项,就像它们是一个单独项一样。

QGraphicsView 是一个窗口部件,这个窗口部件可以显示场景,在需要的情况下提供滚动条,以及影响场景绘制方式的变换能力。这有利于支持缩放和旋转,帮助浏览场景。

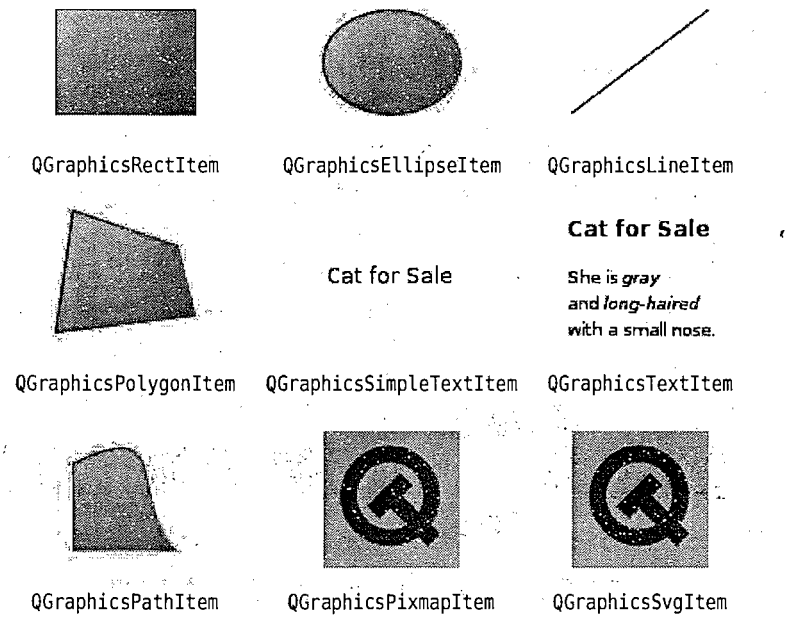


图 8.15 Qt 4.3 中可用的各种图形视图项

默认情况下,QGraphicsView 使用 Qt 的内置二维图形引擎绘图,但这可以改变,在其创建完后调用 setViewport()改为使用 OpenGL 窗口部件。打印一个或部分场景也是很容易的,将在下一节中介绍使用 Qt 打印的几种方法。

这个体系使用三种不同的坐标系统——视口坐标、场景坐标和项坐标——而且还包含从一个坐标系统映射到另一个坐标的函数。视口坐标是 QGraphicsView 的坐标。场景坐标是逻辑坐标,用来布置场景中的项。项坐标针对某一项,并且以(0,0)点为中心。当在场景中移动项时,项坐标保持不变。在实际应用中,我们常常只关心场景坐标(用于布置上层的项),以及项坐标(用于布置子项和绘制项)。依照本身的坐标系统绘制项意味着我们不用去关心项在场景中的位置或者关心需要应用的变换。

视图类用起来很简单,而且具有很强的功能。为了介绍它能做什么,我们来看两个例子。第一个例子是个简单的图表编辑器,我们将看到怎样创建项,以及怎样处理用户交互。第二个例子是个有注解的地图程序,介绍了如何处理大量的图形对象,以及如何以不同的缩放比例高效地绘制它们。

图 8.16 所示的图表应用程序可以让用户创建节点和 Link。节点就是项,是可以在内部显示文本的圆角矩形,而 Link 是连接两个节点的线。被选中的节点用比普通线粗的虚线边缘表示。首先,我们来看看 Link,因为它是最简单的,然后是节点,最后将看到怎样在上下文中使用它们。

```
class Link : public QGraphicsLineItem
{
public:
```

```
    Link(Node *fromNode, Node *toNode);
    ~Link();

    Node *fromNode() const;
    Node *toNode() const;

    void setColor(const QColor &color);
    QColor color() const;

    void trackNodes();

private:
    Node *myFromNode;
    Node *myToNode;
};
```

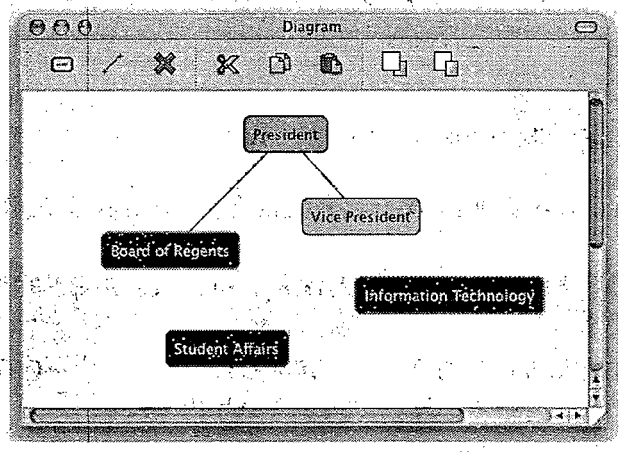


图 8.16 图表应用程序

Link 类派生自 QGraphicsLineItem,它可以在 QGraphicsScene 中显示一条线。Link 有三个属性:连接的两个节点以及画线所用的颜色。我们不需要使用 QColor 成员变量来保存颜色,这是因为简捷的原因。QGraphicsItem 不是 QObject 的子类,但如果需要在 Link 类中添加信号和槽,可以使用 QObject 做多重继承。

在用户拖动一个连接节点到一个不同的位置时,trackNodes()函数用来更新线的端点。

```
Link::Link(Node *fromNode, Node *toNode)
{
    myFromNode = fromNode;
    myToNode = toNode;

    myFromNode->addLink(this);
    myToNode->addLink(this);

    setFlags(QGraphicsItem::ItemIsSelectable);
    setZValue(-1);

    setColor(Qt::darkRed);
    trackNodes();
}
```

当创建一个 Link 时,它会把自己添加到连接的节点中。每个节点保存一个 Link 的集合,而且可以保存任意数量的 Link。图形项有几个标识;但在这种情况下,只需要 Link 可以被选中,从而用户可以选中并删除它。

每个项都有一个(x,y)坐标,以及一个 z 值,指定它在场景中的前后位置。因为我们打算从一个节点的中心向另一个节点的中心画线,所以给直线一个负的 z 值,这样它就会被绘制到所连接的节点的下面。这样,Link 就是它所连接的节点与最近边框之间的线。

在构造函数的末尾，会初始化线的颜色，然后调用 trackNodes()设置线的端点。

```
Link::~Link()
{
    myFromNode->removeLink(this);
    myToNode->removeLink(this);
}
```

当线被销毁时，它将从其连接的节点中删除掉。

```
void Link::setColor(const QColor &color)
{
    setPen(QPen(color, 1.0));
}
```

当设置 Link 的颜色时，只需要用给定的颜色和宽度为 1 的线设置画笔。setPen()函数继承自 QGraphicsLineItem。color()返回画笔的颜色。

```
void Link::trackNodes()
{
    setLine(QLineF(myFromNode->pos(), myToNode->pos()));
}
```

QGraphicsItem::pos()函数返回项相对于场景的位置(针对上层项)，或者相对于父项的位置(针对子项)。

对于 Link 类，可以依赖其父类来处理绘图：QGraphicsLineItem 在场景的两点之间绘制一条直线[用 pen()函数]。

对于 Node 类，将自行处理所有的绘图。节点和 Link 的另外一个不同点是：节点更具交互性。首先来看 Node 的声明，由于它很长，所以把它分成几个片断。

```
class Node : public QGraphicsItem
{
    Q_DECLARE_TR_FUNCTIONS(Node)

public:
    Node();
```

对于 Node 类，我们用 QGraphicsItem 作为基类。Q_DECLARE_TR_FUNCTIONS()宏用来给类添加一个 tr()函数，尽管它不是 QObject 的子类。这种方法简单方便，可以直接使用 tr()，而不是静态的 QObject::tr()或者 QCoreApplication::translate()。

```
    void setText(const QString &text);
    QString text() const;
    void setTextColor(const QColor &color);
    QColor textColor() const;
    void setOutlineColor(const QColor &color);
    QColor outlineColor() const;
    void setBackgroundColor(const QColor &color);
    QColor backgroundColor() const;
```

这些函数只是对私有成员的简单的读取和设置操作。此处提供对文字颜色、节点边缘以及节点背景的控制。

```
    void addLink(Link *link);
    void removeLink(Link *link);
```

就像前面看到的，这些函数被 Link 类调用，会在节点中添加和删除它们自身。

```
    QRectF boundingRect() const;
    QPainterPath shape() const;
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option, QWidget *widget);
```

当创建 QGraphicsItem 的子类时，要想自己实现绘图，一般是重新实现 boundingRect()和 paint()。

如果不重新实现 shape(),基类的实现将会退而使用 boundingRect()。在这种情况下,我们重新实现了 shape()将节点的圆角考虑进去,从而返回更准确的形状。

视图体系用外接矩形来决定一个项是否需要被绘制。这使得 QGraphicsView 可以很迅速地显示任意大的场景,尽管此时只有一小部分项是可见的。shape 用来决定一个点是否在项内,或者是否两个项是重合的。

```
protected:
    void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event);
    QVariant itemChange(GraphicsItemChange change,
                        const QVariant &value);
```

在图表应用程序中将提供一个 Properties 对话框,用来编辑节点的位置、颜色和文字。为了更加方便,将允许用户通过双击节点修改文字。

如果节点被移动了,必须确保与其相连的 Link 做相应的更新。我们重新实现 itemChange()处理器来做更新,每当项的属性(包括其位置)改变时被调用。我们不用 mouseMoveEvent()事件达到这一目的,是因为当节点被程式化地移动时,它不会被调用。

```
private:
    QRectF outlineRect() const;
    int roundness(double size) const;

    QSet<Link *> myLinks;
    QString myText;
    QColor myTextColor;
    QColor myBackgroundColor;
    QColor myOutlineColor;
};
```

outlineRect()私有函数可以返回由节点绘制的矩形,而 roundness()可以返回一个基于矩形宽度和高度的合适的圆度系数。

如同 Link 要保存它所连接的节点的信息一样,Node 也会保存它的 Link 信息。当一个节点被删除了,所有与该节点相连的 Link 也会被删除。

现在来看 Node 类的实现,首先从其构造函数开始。

```
Node::Node()
{
    myTextColor = Qt::darkGreen;
    myOutlineColor = Qt::darkBlue;
    myBackgroundColor = Qt::white;

    setFlags(ItemIsMovable | ItemIsSelectable);
}
```

我们初始化颜色,设置节点项可被移动和选取。z 的值被默认地设置为 0,允许调用者设置节点在场景中的位置。

```
Node::~Node()
{
    foreach (Link *link, myLinks)
        delete link;
}
```

析构函数删除所有节点的 Link。当一个 Link 被销毁了,它将把自己从其连接的节点中删除。我们迭代(复制)Link 的集合,而不是使用 qDeleteAll()来避免副作用,因为 Link 的集合被 Link 的析构函数间接地访问。

```
void Node::setText(const QString &text)
{
    prepareGeometryChange();
    myText = text;
    update();
}
```

无论何时修改了项,都会影响到它的显示,所以必须调用 update()来安排一个重绘。例如项的外接矩形可能会改变(因为新的文字可能比现在的文字短或者长),必须在做影响项的外接矩形的修改之前立即调用 prepareGeometryChange()。

我们将跳过 text()、textColor()、outlineColor()和 backgroundColor()提取函数,因为它们只是简单地返回其对应的私有变量。

```
void Node::setTextColor(const QColor &color)
{
    myTextColor = color;
    update();
}
```

当设置文字的颜色时,必须调用 update()来安排一个重绘,以便使用新的颜色绘制项。没有必要调用 prepareGeometryChange(),因为项的大小不会受颜色改变的影响。我们将忽略边缘和背景颜色的设置函数,因为它们在结构上与这个函数是相同的。

```
void Node::addLink(Link *link)
{
    myLinks.insert(link);
}

void Node::removeLink(Link *link)
{
    myLinks.remove(link);
}
```

这里只是简单地向节点的 Link 的集合中添加和删除给定的 Link。

```
QRectF Node::outlineRect() const
{
    const int Padding = 8;
    QFontMetricsF metrics = qApp->font();
    QRectF rect = metrics.boundingRect(myText);
    rect.adjust(-Padding, -Padding, +Padding, +Padding);
    rect.translate(-rect.center());
    return rect;
}
```

我们使用这个私有函数来计算一个包围节点文字的矩形,该矩形有 8 像素的边距。由字体规格函数返回的外接矩形总是把(0,0)作为其左上角。因为我们想把项的中心点作为文字的居中点,因此移动这个矩形,使它的中心在(0,0)。

尽管是以像素为单位来进行思考和计算的,但它却是场景中名义上的单位。场景(或者父项)可能被缩放、旋转、拉伸或者只是受反走样的影响,因而场景上实际的像素数可能有所不同。

```
QRectF Node::boundingRect() const
{
    const int Margin = 1;
    return outlineRect().adjusted(-Margin, -Margin, +Margin, +Margin);
}
```

boundingRect()函数会由 QGraphicsView 调用,以决定是否需要绘制项。我们使用边缘矩形,但留些额外的边白,因为如果需要绘制边缘,由这个函数返回的矩形必须留出至少半个画笔宽的距离。

```
QPainterPath Node::shape() const
{
    QRectF rect = outlineRect();

    QPainterPath path;
    path.addRoundRect(rect, roundness(rect.width()),
                      roundness(rect.height()));
    return path;
}
```

shape()函数可以由 QGraphicsView 调用,用来做精确的碰撞检测。通常,可以忽略它,由项基于外接矩形自行计算形状。这里重新实现了它,由其返回一个 QPainterPath 对象,该对象代表了一个圆角矩形。因此,如果点击圆角矩形外、外接矩形内的区域则不会选中项。

当创建圆角矩形时,可以传入一个可选的参数来指定圆角率。利用私有函数 roundness()可以计算出一个合适的值。

```
void Node::paint(QPainter *painter,
                 const QStyleOptionGraphicsItem *option,
                 QWidget * /* widget */)
{
    QPen pen(myOutlineColor);
    if (option->state & QStyle::State_Selected) {
        pen.setStyle(Qt::DotLine);
        pen.setWidth(2);
    }
    painter->setPen(pen);
    painter->setBrush(myBackgroundColor);

    QRectF rect = outlineRect();
    painter->drawRoundRect(rect, roundness(rect.width()),
                           roundness(rect.height()));

    painter->setPen(myTextColor);
    painter->drawText(rect, Qt::AlignCenter, myText);
}
```

paint()函数就是绘制项的地方。如果项被选中,就将画笔的风格改为点线型,而且使线更粗些;否则,默认使用像素为 1 的实线。还设置了画刷的颜色为背景色。

然后,绘制一个与边缘矩形大小相等的圆角矩形,其中使用了由 roundness()私有函数返回的圆角率。最后,在圆角矩形的上面绘制文字,使其居中于边缘矩形。

QStyleOptionGraphicsItem 类型的 option 参数是 Qt 的一个不寻常的类,因为它提供了几个公有成员变量。这包括当前的布局方向、字体规格、调色板、矩形、状态(如选中、"获取焦点",以及其他一些状态等)、变换矩阵和细节级别。这里我们检查了状态变量来确定节点是否被选中。

```
QVariant Node::itemChange(GraphicsItemChange change,
                          const QVariant &value)
{
    if (change == ItemPositionHasChanged) {
        foreach (Link *link, myLinks)
            link->trackNodes();
    }
    return QGraphicsItem::itemChange(change, value);
}
```

一旦用户拖动一个节点,就会调用 itemChange()处理器,由 ItemPositionHasChanged 作为第一个参数。为了保证 Link 位于正确的位置,可以遍历节点的 Link 集合,通知每一条线更新它们的端点。最后,调用基类的实现以确保基类也得到了通知。

```
void Node::mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event)
{
    QString text = QInputDialog::getText(event->widget(),
                           tr("Edit Text"), tr("Enter new text:"),
                           QLineEdit::Normal, myText);
    if (!text.isEmpty())
        setText(text);
}
```

如果用户双击该节点,就弹出一个显示当前文字的对话框,并且允许用户修改它。如果用户点击了 Cancel,将会返回一个空字符串。因此,只有当字符串不为空时,这个修改才会有效。稍后将看到怎样修改其他的节点属性。

```
int Node::roundness(double size) const
{
    const int Diameter = 12;
    return 100 * Diameter / int(size);
}
```

roundness()函数返回合适的圆角率,以确保节点的转角是直径为12的四分之一圆。圆角率的范围必须在0(直角形)到99(满圆形)之间。

我们已经知道了两个自定义项的类的实现,现在是介绍如何使用它们的时候了。图表应用程序是一个标准的带菜单和工具栏的主窗口应用程序。我们不会查看所有的实现细节,而是专注于与视图体系相关的细节。首先来看从QMainWindow子类定义中截取的一部分。

```
class DiagramWindow : public QMainWindow
{
    Q_OBJECT

public:
    DiagramWindow();

private slots:
    void addNode();
    void addLink();
    void del();
    void cut();
    void copy();
    void paste();
    void bringToFront();
    void sendToBack();
    void properties();
    void updateActions();

private:
    typedef QPair<Node *, Node *> NodePair;

    void createActions();
    void createMenus();
    void createToolBars();
    void setZValue(int z);
    void setupNode(Node *node);
    Node *selectedNode() const;
    Link *selectedLink() const;
    NodePair selectedNodePair() const;

    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *editToolBar;
    QAction *exitAction;
    ...
    QAction *propertiesAction;

    QGraphicsScene *scene;
    QGraphicsView *view;

    int minZ;
    int maxZ;
    int seqNumber;
};
```

大多数私有槽的用途都应当可以清楚地从它们的名字中看出来。properties()槽用来在节点被选中时弹出一个属性对话框,或者在Link被选中时弹出一个QColorDialog对话框。updateActions()槽可以根据被选中的项来激活或取消操作。

```
DiagramWindow::DiagramWindow()
{
    scene = new QGraphicsScene(0, 0, 600, 500);
```

```
    view = new QGraphicsView;
    view->setScene(scene);
    view->setDragMode(QGraphicsView::RubberBandDrag);
    view->setRenderHints(QPainter::Antialiasing
                         | QPainter::TextAntialiasing);
    view->setContextMenuPolicy(Qt::ActionsContextMenu);
    setCentralWidget(view);

    minZ = 0;
    maxZ = 0;
    seqNumber = 0;

    createActions();
    createMenus();
    createToolBars();

    connect(scene, SIGNAL(selectionChanged()),
            this, SLOT(updateActions()));

    setWindowTitle(tr("Diagram"));
    updateActions();
}
```

我们从创建一个图形场景开始,其原点为(0,0),宽为 600,高为 500。然后,创建一个视图来显示场景。在下一个例子中,将用 QGraphicsView 的子类的方式代替直接使用 QGraphicsView 来自定义其行为。

通过单击,可以选中那些可选项。要想一次同时选中多个项,用户可以在按住 Ctrl 键时单击那些项。把拖动模式设置为 QGraphicsView::RubberBandDrag,则意味着用户可以通过圈选选中它们。

minZ 和 maxZ 数字用在 sendToBack()和 bringToFront()函数中。序号用来为用户添加的每个节点提供一个唯一的初始化文字。

信号-槽连接可以确保场景的选取一旦发生改变,就可以启用或者禁用应用程序的那些动作,以便只让那些有意义的操作才可用。我们调用 updateActions()来设置操作的初始可用状态。

```
void DiagramWindow::addNode()
{
    Node *node = new Node;
    node->setText(tr("Node %1").arg(seqNumber + 1));
    setupNode(node);
}
```

当用户添加了一个新的节点时,就可以创建一个新的 Node 类的实例,并赋予它默认的文字,然后把节点传给 setupNode()来定位和选取它。我们使用一个单独的函数来完成一个节点的添加工作,因为在实现 paste()时还需要用到这一功能。

```
void DiagramWindow::setupNode(Node *node)
{
    node->setPos(QPoint(80 + (100 * (seqNumber % 5)),
                        80 + (50 * ((seqNumber / 5) % 7))));
    scene->addItem(node);
    ++seqNumber;

    scene->clearSelection();
    node->setSelected(true);
    bringToFront();
}
```

此函数可以在场景中定位一个新近添加或者复制的节点。序列号的用途可以确保新节点被添加到一个不同的位置,而不是放在彼此之上。我们清除了当前的选中操作,并且将新添加的节点选中。bringToFront()调用确保了新节点比任何其他节点都靠前。

```
void DiagramWindow::bringToFront()
{
    ++maxZ;
    setZValue(maxZ);
}
void DiagramWindow::sendToBack()
{
    --minZ;
    setZValue(minZ);
}

void DiagramWindow::setZValue(int z)
{
    Node *node = selectedNode();
    if (node)
        node->setZValue(z);
}
```

bringToFront()槽会增加 maxZ 的值,然后会把当前选中的节点的 z 值设置为 maxZ。sendToBack()槽会使用 minZ,并且具有相反的效果。这两个值都是在 setZValue()私有函数中定义的。

```
Node *DiagramWindow::selectedNode() const
{
    QList<QGraphicsItem *> items = scene->selectedItems();
    if (items.count() == 1) {
        return dynamic_cast<Node *>(items.first());
    } else {
        return 0;
    }
}
```

场景中所有被选中的节点的列表可以通过调用 QGraphicsScene::selectedItems()获得。如果仅有一个节点被选中,selectedNode()函数被用来返回单个节点;如果没有节点被选中,则会返回一个空指针。如果确实存在一个被选中的项,若该项是一个节点,这个类型强制转换将返回一个 Node 指针,若是一个 Link,则会返回一个空指针。

还有一个 selectedLink()函数,如果确实存在一个选中的项并且该项是一个 Link,它也会返回一个选中的 Link 项指针。

```
void DiagramWindow::addLink()
{
    NodePair nodes = selectedNodePair();
    if (nodes == NodePair())
        return;

    Link *link = new Link(nodes.first, nodes.second);
    scene->addItem(link);
}
```

如果有两个节点被选中,用户就可以添加一个 Link。如果 selectedNodePair()函数返回两个被选中的节点,则创建一个新节点。Link 的构造函数将直线的端点设置为从第一个节点的中心到第二个节点的中心。

```
DiagramWindow::NodePair DiagramWindow::selectedNodePair() const
{
    QList<QGraphicsItem *> items = scene->selectedItems();
    if (items.count() == 2) {
        Node *first = dynamic_cast<Node *>(items.first());
        Node *second = dynamic_cast<Node *>(items.last());
        if (first && second)
            return NodePair(first, second);
    }
    return NodePair();
}
```

此函数类似于以前看到的 selectedNode()。如果确实存在两个被选中的项,并且它们都是节点,那么它们都将会返回;否则,将只会返回一对空指针。

```
void DiagramWindow::del()
{
    QList<QGraphicsItem *> items = scene->selectedItems();
    QMutableListIterator<QGraphicsItem *> i(items);
    while (i.hasNext()) {
        Link *link = dynamic_cast<Link *>(i.next());
        if (link) {
            delete link;
            i.remove();
        }
    }

    qDeleteAll(items);
}
```

无论选中的项是节点、Link 或者是既有节点又有 Link,该槽都可以删除这些选中的任意项。当删除一个节点时,它的析构函数就会删除与其关联的所有 Link。为了避免 Link 的二次删除,可以在删除节点之前先删除那些 Link 项。

```
void DiagramWindow::properties()
{
    Node *node = selectedNode();
    Link *link = selectedLink();

    if (node) {
        PropertiesDialog dialog(node, this);
        dialog.exec();
    } else if (link) {
        QColor color = QColorDialog::getColor(link->color(), this);
        if (color.isValid())
            link->setColor(color);
    }
}
```

如果用户触发了 Properties 动作,并且只有一个节点被选中,就可以调用 Properties 对话框。这个对话框允许用户修改节点的文字、位置和颜色。因为 PropertiesDialog 可以直接操作一个节点的指针,我们只需将其执行为模态对话框即可,它会自行照料自己的。

如果一个 Link 被选中,就可以使用 Qt 内置的 QColorDialog::getColor()静态简便函数弹出一个颜色对话框。如果用户选择了一种颜色,就将它设置为 Link 的颜色。

如果节点的属性或者 Link 的颜色发生了改变,那么这些改变就可以通过设置的那些函数生效,它们会调用 update()来确保节点和 Link 使用自己的新设置来重绘自己。

在这样的应用程序中,用户常常希望能够使用剪切、复制和粘贴项,而能够支持这些功能的办法就是以文本的方式来显示这些项,这一点会在讨论相关代码时看到。我们只处理这些节点,是因为复制或者粘贴 Link 并没有什么意义,它们只存在于各个节点之间。

```
void DiagramWindow::cut()
{
    Node *node = selectedNode();
    if (!node)
        return;

    copy();
    delete node;
}
```

Cut 操作是两部分操作:把被选中的项复制到剪贴板,然后再将该项删除掉。复制使用 copy()槽实现,该槽用于 Copy 操作,且删除使用 C++ 的标准 delete 操作符,依靠节点的析构函数可以删除与节点相连的 Link,并且从场景中删除该节点。

```
void DiagramWindow::copy()
{
    Node *node = selectedNode();
    if (!node)
        return;

    QString str = QString("Node %1 %2 %3 %4")
                  .arg(node->textColor().name())
                  .arg(node->outlineColor().name())
                  .arg(node->backgroundColor().name())
                  .arg(node->text());
    QApplication::clipboard()->setText(str);
}
```

QColor::name()函数返回一个QString,包含一个HTML风格的颜色字符串,格式为"#RRGGBB",每个颜色元素由一个从0x00到0xFF(0到255)的十六进制值来表示。我们把字符串写到剪贴板中,该字符串是一个以"Node"开始的单行文字,接着是节点的三个颜色值,最后还有节点的文字用空格彼此隔开。例如:

```
Node #aa0000 #000080 #ffffff Red herring
```

这段文字可以由paste()函数解析:

```
void DiagramWindow::paste()
{
    QString str = QApplication::clipboard()->text();
    QStringList parts = str.split(" ");
    if (parts.count() >= 5 && parts.first() == "Node") {
        Node *node = new Node;
        node->setText(QStringList(parts.mid(4)).join(" "));
        node->setTextColor(QColor(parts[1]));
        node->setOutlineColor(QColor(parts[2]));
        node->setBackgroundColor(QColor(parts[3]));
        setupNode(node);
    }
}
```

我们把剪贴板中的文字拆分到一个QStringList中。使用前面的例子,将得到列表["Node","#aa0000","#000080","#ffffff","Red","herring"]。要成为一个有效的节点,列表中就必须至少存在5个元素:单词"Node"、三个颜色值,以及至少一个单词作为文字。如果情况确实如此,就可以创建一个新的节点,设置显示的文字为以空格分隔符串联的字符串的第5个单词以及其后面的单词的组合。我们设置颜色值为串联字符串的第2、第3和第4部分,使用QColor的构造函数,该函数能够接受由QColor::name()函数返回的颜色名称。

为了保证完整性,下面给出的是updateActions()槽,它用来启用或者禁用Edit菜单以及上下文菜单中的那些操作。

```
void DiagramWindow::updateActions()
{
    bool hasSelection = !scene->selectedItems().isEmpty();
    bool isNode = (selectedNode() != 0);
    bool isNodePair = (selectedNodePair() != NodePair());

    cutAction->setEnabled(isNode);
    copyAction->setEnabled(isNode);
    addLinkAction->setEnabled(isNodePair);
    deleteAction->setEnabled(hasSelection);
    bringToFrontAction->setEnabled(isNode);
    sendToBackAction->setEnabled(isNode);
    propertiesAction->setEnabled(isNode);

    foreach (QAction *action, view->actions())
        view->removeAction(action);
```

```
    foreach (QAction *action, editMenu->actions()) {
        if (action->isEnabled())
            view->addAction(action);
    }
}
```

现在,就完成了对图表应用程序的介绍,可以将注意力转移到第二个图形视图方面的例子上,即 Cityscape。

如图 8.17 所示,Cityscape 应用程序显示了城市中的建筑物、街区和公园的虚拟地图,最为重要的那些建筑会用它们的名字加以标注。它可以让用户通过鼠标和键盘来移动和缩放地图。首先介绍 Cityscape 类,它提供了应用程序的主窗口。

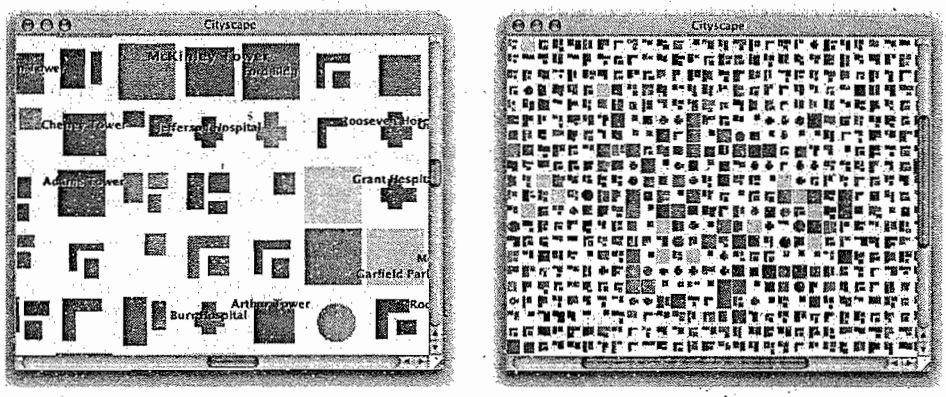


图 8.17 具有两种不同显示比例的 Cityscape 应用程序

```
class Cityscape : public QMainWindow
{
    Q_OBJECT

public:
    Cityscape();

private:
    void generateCityBlocks();

    QGraphicsScene *scene;
    CityView *view;
};
```

此应用程序没有菜单栏和工具栏,它只是简单地使用 CityView 窗口部件来显示有标注的地图。CityView 类从 QGraphicsView 类派生过来。

```
Cityscape::Cityscape()
{
    scene = new QGraphicsScene(-22.25, -22.25, 1980, 1980);
    scene->setBackgroundBrush(QColor(255, 255, 238));
    generateCityBlocks();

    view = new CityView;
    view->setScene(scene);
    setCentralWidget(view);

    setWindowTitle(tr("Cityscape"));
}
```

构造函数创建了一个 QGraphicsScene,调用 generateCityBlocks()生成一幅地图。这幅地图包含 2000 个街区和 200 个标注。

首先将看到的是图形项的子类 CityBlock,然后是图形项的子类 Annotation,最后是图形视图的子类 CityView。

```
class CityBlock : public QGraphicsItem
{
public:
    enum Kind { Park, SmallBuilding, Hospital, Hall, Building, Tower,
                LShapedBlock, LShapedBlockPlusSmallBlock, TwoBlocks,
                BlockPlusTwoSmallBlocks };

    CityBlock(Kind kind);

    QRectF boundingRect() const;
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option, QWidget *widget);

private:
    int kind;
    QColor color;
    QPainterPath shape;
};
```

一个城市的街区会有类型、颜色和形状属性。因为城市街区是不可选的,所以不用担心像前面例子中 Node 类那样去重新实现 shape()函数。

```
CityBlock::CityBlock(Kind kind)
{
    this->kind = kind;

    int green = 96 + (std::rand() % 64);
    int red = 16 + green + (std::rand() % 64);
    int blue = 16 + (std::rand() % green);
    color = QColor(red, green, blue);

    if (kind == Park) {
        color = QColor(192 + (std::rand() % 32), 255,
                       192 + (std::rand() % 16));
        shape.addRect(boundingRect());
    } else if (kind == SmallBuilding) {
        ...
    } else if (kind == BlockPlusTwoSmallBlocks) {
        int w1 = (std::rand() % 10) + 8;
        int h1 = (std::rand() % 28) + 8;
        int w2 = (std::rand() % 10) + 8;
        int h2 = (std::rand() % 10) + 8;
        int w3 = (std::rand() % 6) + 8;
        int h3 = (std::rand() % 6) + 8;
        int y = (std::rand() % 4) - 16;
        shape.addRect(QRectF(-16, -16, w1, h1));
        shape.addRect(QRectF(-16 + w1 + 4, y, w2, h2));
        shape.addRect(QRectF(-16 + w1 + 4,
                             y + h2 + 4 + (std::rand() % 4), w3, h3));
    }
}
```

构造函数设置了一种随机颜色,可以根据需要显示的街区的类型生成一个合适的 QPainterPath。

```
QRectF CityBlock::boundingRect() const
{
    return QRectF(-20, -20, 40, 40);
}
```

每个街区都会占用一个 40×40 的正方形空格,其中心点是(0,0)。

```
void CityBlock::paint(QPainter *painter,
                      const QStyleOptionGraphicsItem *option,
                      QWidget * /* widget */)
{
    if (option->levelOfDetail < 4.0) {
        painter->fillPath(shape, color);
    } else {
        QLinearGradient gradient(QPoint(-20, -20), QPoint(+20, +20));
```

```
        int coeff = 105 + int(std::log(option->levelOfDetail - 4.0));
        gradient.setColorAt(0.0, color.lighter(coeff));
        gradient.setColorAt(1.0, color.darker(coeff));
        painter->fillPath(shape, gradient);
    }
}
```

在 paint()函数中,我们用给定的 QPainter 绘制图形。需要区分两种情况:

- 如果缩放因子比 4.0 小,则使用纯色填充图形。
- 如果缩放因子是 4.0 或者更大,使用 QLinearGradient 填充图形,以产生一种奇妙的光照效果。

QStyleOptionGraphicsItem 类的 levelOfDetail 成员函数存储一个浮点值,该值就是缩放因子。1.0 意味着以原始大小显示场景,0.5 意味着以原始大小的一半显示场景,2.5 表示显示的大小是原始尺寸的 2.5 倍。使用"详细程度"信息可以让我们在那些放得太大以至于不能显示细节的场景中使用更快的绘图算法。

CityBlock 类很好用,但事实上,当场景缩放时造成项也被缩放,这样就给显示文字的那些项带来了问题。一般情况下,我们不希望文字随着场景缩放。视图体系为这一问题提供了通用的解决方式,即使用 ItemIgnoresTransformations 标识。Annotation 类中就使用了此标识:

```
class Annotation : public QGraphicsItem
{
public:
    Annotation(const QString &text, bool major = false);

    void setText(const QString &text);
    QString text() const;

    QRectF boundingRect() const;
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option, QWidget *widget);

private:
    QFont font;
    QString str;
    bool major;
    double threshold;
    int y;
};
```

构造函数使用文字以及称作 major 的布尔型标识作为参数,这规定了该注解是主注解还是辅注解,还将影响到字体的大小。

```
Annotation::Annotation(const QString &text, bool major)
{
    font = qApp->font();
    font.setBold(true);
    if (major) {
        font.setPointSize(font.pointSize() + 2);
        font.setStretch(QFont::SemiExpanded);
    }

    if (major) {
        threshold = 0.01 * (40 + (std::rand() % 40));
    } else {
        threshold = 0.01 * (100 + (std::rand() % 100));
    }

    str = text;
    this->major = major;
    y = 20 - (std::rand() % 40);

    setZValue(1000);
    setFlag(ItemIgnoresTransformations, true);
}
```

在构造函数中，假设主注解代表一个重要的建筑物或者地标，则首先可以把它设置成较大而且加粗的字体。阈值是通过伪随机方法计算出来的，比它小的注解不会被显示出来。主注解有较小的阈值，当场景缩小时，不太重要的注解会消失。

我们设置 z 的值为 1000，以确保注解显示在最上面，而且使用 ItemIgnoresTransformations 标识来确保无论场景被如何缩放，注解的大小都可以保持不变。

```
void Annotation::setText(const QString &text)
{
    prepareGeometryChange();
    str = text;
    update();
}
```

如果注解的文字需要修改，它可能比以前更长或者更短了，因此必须通知视图体系修改该项的尺寸。

```
QRectF Annotation::boundingRect() const
{
    QFontMetricsF metrics(font);
    QRectF rect = metrics.boundingRect(str);
    rect.moveCenter(QPointF(0, y));
    rect.adjust(-4, 0, +4, 0);
    return rect;
}
```

我们从注解的字体中取得字体规格信息，使用它们计算出文字的外接矩形。然后移动矩形的中心点到注解的 y 值的偏移位置，并且使矩形稍微大一些。外接矩形左边和右边的额外像素使文字相对于边框还有一些空间。

```
void Annotation::paint(QPainter *painter,
                       const QStyleOptionGraphicsItem *option,
                       QWidget * /* widget */)
{
    if (option->levelOfDetail <= threshold)
        return;

    painter->setFont(font);

    QRectF rect = boundingRect();

    int alpha = int(30 * std::log(option->levelOfDetail));
    if (alpha >= 32)
        painter->fillRect(rect, QColor(255, 255, 255, qMin(alpha, 63)));

    painter->setPen(Qt::white);
    painter->drawText(rect.translated(+1, +1), str,
                      QTextOption(Qt::AlignCenter));
    painter->setPen(Qt::blue);
    painter->drawText(rect, str, QTextOption(Qt::AlignCenter));
}
```

如果场景缩小的尺寸超过了注解的阈值，就不再绘制该注解。如果场景被放得足够大，就事先绘制一个半透明的白色矩形，这有助于在黑色的块上显示文字。

我们绘制文字两次，一次用白色，一次用蓝色。白色的文字在水平和竖直方向上平移一个像素来创建一个阴影效果，使文字更容易被读出来。

介绍完了块和注解是怎样完成的之后，下面继续介绍 Cityscape 应用程序的最后一个方面，即用户自定义的 QGraphicsView 子类：

```
class CityView : public QGraphicsView
{
    Q_OBJECT
```

```
public:
    CityView(QWidget *parent = 0);
protected:
    void wheelEvent(QWheelEvent *event);
};
```

默认情况下,QGraphicsView 类可以在需要的时候自动显示一些滚动条,但没有提供任何缩放场景的方法。因此,我们创建了微型 CityView 子类,以提供用户使用鼠标缩放的功能。

```
CityView::CityView(QWidget *parent)
    : QGraphicsView(parent)
{
    setDragMode(ScrollHandDrag);
}
```

设置拖动模式可支持通过鼠标拖动来滚动屏幕。

```
void CityView::wheelEvent(QWheelEvent *event)
{
    double numDegrees = -event->delta() / 8.0;
    double numSteps = numDegrees / 15.0;
    double factor = std::pow(1.125, numSteps);
    scale(factor, factor);
}
```

当用户滚动鼠标滚轴,就会触发滚轴事件。我们需要简单地计算出一个适当的比例因子,然后调用 QGraphicsView::scale()。此数学公式有点巧妙,基本上是按滚轴步长的 1.125 倍上下移动场景。

这样就完成了两个图形视图的例子。Qt 的图形视图体系内容很多,因此还有很多这里没有篇幅去介绍的。比如支持拖动和释放操作,图形项可以有提示和自定义的光标。动画效果可以通过几种方式实现——例如,在希望显示动画的项上使用 QGraphicsItemAnimations,使用 QTimeLine 播放动画。也可以通过继承 QObject(应用多继承)创建图形项的子类,重新实现 QObject::timeEvent()显示动画。

8.5 打印

Qt 中的打印与在 QWidget、QPixmap 或者 QImage 上的绘制非常相似。它包括以下步骤:

1. 创建一个当作绘制设备的 QPrinter。
2. 弹出一个 QPrintDialog 对话框,以允许用户选择打印机并且设置一些选项。
3. 创建一个在 QPrinter 上操作的 QPainter。
4. 使用 QPainter 绘制一页。
5. 调用 QPainter::newPage()来进行下一页的绘制。
6. 重复步骤 4 和步骤 5,直到所有页都被打印为止。

在 Windows 和 Mac OS X 上,QPrinter 会使用系统的打印机驱动程序。在 UNIX 上,它会产生 PostScript 并且把它发送给 lp 或者 lpr 命令[或者是由 QPrinter::setPrintProgram()设置的其他程序]。QPrinter 也可以通过调用 setOutputFormat(QPrinter::PdfFormat)来生成 PDF 文件①。

我们先从一个只有单页打印的简单实例开始。这里的第一个例子可以打印一个 QImage,如图 8.18 所示。

```
void PrintWindow::printImage(const QImage &image)
{
    QPrintDialog printDialog(&printer, this);
```

① 有望在 Qt 4.4 中引入一些打印预览类。

```
    if (printDialog.exec()) {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        QSize size = image.size();
        size.scale(rect.size(), Qt::KeepAspectRatio);
        painter.setViewport(rect.x(), rect.y(),
                            size.width(), size.height());
        painter.setWindow(image.rect());
        painter.drawImage(0, 0, image);
    }
}
```

假设 PrintWindow 类已经有了类型为 QPrinter、名字为 printer 的一个成员变量。可以简单地在 printImage()的栈里创建 QPrinter,但是在打印完之后再打印另一个的时候,将不会记得用户的设置。

可以创建一个 QPrintDialog,并且调用 exec()来显示它。如果用户单击 OK 按钮,它就返回 true;否则,它就返回 false。在调用 exec()之后,QPrinter 对象就准备好了。(不调用 QPrintDialog 也是可以的,可以通过直接调用 QPrinter 的成员函数来设置它。)

接下来创建一个 QPainter,在 QPainter 上进行绘制。设置窗口为图片大小的矩形,并且视口也为同样比例大小的矩形,并在位置(0,0)处绘制一个图片。

默认情况下,QPainter 的窗口会被初始化,这样就可以让打印机和屏幕具有相似的分辨率(通常是每英寸[①] 72 到 100 点之间的某个值),这样使得把窗口绘制代码重新使用到打印中变得非常简单。这里不用考虑此问题,因为我们会自行设置自己的窗口。

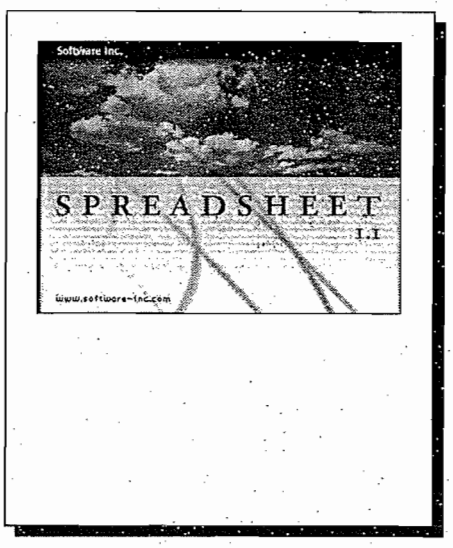


图 8.18 打印 QImage

在本例中,我们选择打印一个图片,但打印图形视图场景也同样很简单。为了打印整个场景,可以调用 QGraphicsScene::render()或者 QGraphicsView::render(),QPainter 作为第一个参数。如果只想打印部分场景,可以使用 render()函数的可选参数指定绘制的目标矩形(在页的什么地方绘制场景),以及源矩形(需要绘制的部分)。

打印不超过一页的项是非常简单的,但是很多应用程序通常都需要打印多页。对于这些情况,需要一次绘制一页并且调用 newPage()来前进到下一页。这将会导致一个问题,也就是如何决定可以在一页上打印多少信息。在 Qt 中,有两种方式处理多页文档:

- 可以把数据转换为 HTML,并且使用 Qt 的富文本引擎 QTextDocument 进行显示。
- 可以执行绘制并且手动分页。

下面将依次讲述这两种方法。作为一个实例,我们将会打印一个花的说明:一个用文字描述花的名称的列表。在这个说明中的每一项都存储了一个格式为"名称:描述"的字符串,例如:

```
Miltonopsis santanae: A most dangerous orchid species.
```

因为每一种花的数据都由一个单一字符串进行表示,所以可以使用一个 QStringList 来代表这个说明中的所有花。在这里使用 Qt 的富文本引擎来打印一个花的说明:

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
    QString html;
```

① 1 英寸 = 2.54 cm。

```
        foreach (QString entry, entries) {
            QStringList fields = entry.split(": ");
            QString title = Qt::escape(fields[0]);
            QString body = Qt::escape(fields[1]);
            html += "<table width=\"100%\" border=1 cellspacing=0>\n"
                    "<tr><td bgcolor=\"lightgray\"><font size=\"+1\">"
                    "<b><i>" + title + "</i></b></font>\n<tr><td>" + body
                    + "\n</table>\n<br>\n";
        }
        printHtml(html);
    }
```

第一步是把 QStringList 转换为 HTML。每种花都变成了一个 HTML 表格中的两个单元。我们使用 Qt::escape()来把特殊字符“&”、“<”、“>”替换为相应的 HTML 项(“&”,“<”,“>”),然后调用 printHtml()打印这个文本。

```
void PrintWindow::printHtml(const QString &html)
{
    QPrintDialog printDialog(&printer, this);
    if (printDialog.exec()) {
        QTextDocument textDocument;
        textDocument.setHtml(html);
        textDocument.print(&printer);
    }
}
```

printHtml()函数弹出一个 QPrintDialog,负责打印 HTML 文档。它可以在 Qt 的应用程序中不做改变地重新使用来打印任意的 HTML 页面。打印页如图 8.19 所示。

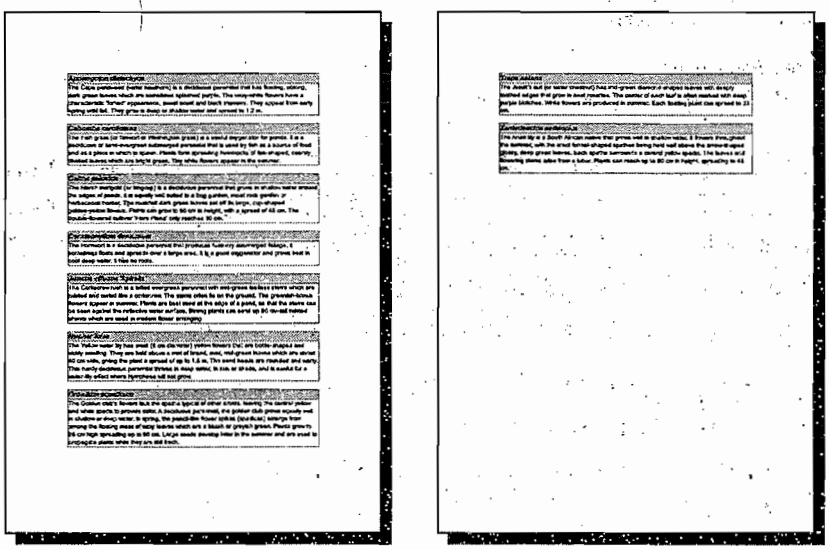


图 8.19 使用 QTextDocument 打印花的说明

转换文档为 HTML 并使用 QTextDocument 打印它,是目前为止打印报告和其他一些复杂文档的最方便的选择。如果需要更多的可控性,可以手动地控制页面的布局和绘制。让我们来看看如何使用这个方法打印花的说明。以下是新的 printFlowerGuide()函数:

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
    QPrintDialog printDialog(&printer, this);
    if (printDialog.exec()) {
        QPainter painter(&printer);
        QList<QStringList> pages;
```

```
        paginate(&painter, &pages, entries);
        printPages(&painter, pages);
    }
}
```

在设置完打印机并且构造完绘图器后，就可以调用 paginate()帮助函数来决定哪一个条目将要在哪一页出现。这样做的结果就放在一个 QStringList 的列表中，每一个 QStringList 保存的是一页中的所有条目。我们将此结果传给 printPages()。

例如，假设花的说明包含 6 个条目，分别是 A，B，C，D，E 和 F。现在假设 A 和 B 在第 1 页；C，D 和 E 在第 2 页；F 在第 3 页。pages 列表将会是这样的：0 位置是一个[A，B]列表，1 位置是一个[C，D，E]列表，而 2 位置是一个[F]列表。

```
void PrintWindow::paginate(QPainter *painter, QList<QStringList> *pages,
                           const QStringList &entries)
{
    QStringList currentPage;
    int pageHeight = painter->window().height() - 2 * LargeGap;
    int y = 0;

    foreach (QString entry, entries) {
        int height = entryHeight(painter, entry);
        if (y + height > pageHeight && !currentPage.empty()) {
            pages->append(currentPage);
            currentPage.clear();
            y = 0;
        }
        currentPage.append(entry);
        y += height + MediumGap;
    }
    if (!currentPage.empty())
        pages->append(currentPage);
}
```

paginate()函数把花的说明的条目分配到页中。它依赖于 entryHeight()函数，后者计算了一个条目的高度。它也把页面顶部和底部页眉和页脚也计算进去了，也就是 LargeGap 的大小。

我们遍历这些条目并且把它们添加到当前页中，直到已经无法在当前页放下条目为止。然后，把当前页添加到 pages 列表中并且开始新的一页。

```
int PrintWindow::entryHeight(QPainter *painter, const QString &entry)
{
    QStringList fields = entry.split(": ");
    QString title = fields[0];
    QString body = fields[1];

    int textWidth = painter->window().width() - 2 * SmallGap;
    int maxHeight = painter->window().height();

    painter->setFont(titleFont);
    QRect titleRect = painter->boundingRect(0, 0, textWidth, maxHeight,
                                            Qt::TextWordWrap, title);
    painter->setFont(bodyFont);
    QRect bodyRect = painter->boundingRect(0, 0, textWidth, maxHeight,
                                           Qt::TextWordWrap, body);
    return titleRect.height() + bodyRect.height() + 4 * SmallGap;
}
```

entryHeight()函数使用 QPainter::boundingRect()来计算一个条目所需的垂直空间。图 8.20 显示了花的条目的布局并且也说明了 SmallGap 和 MediumGap 常量的意义。

```
void PrintWindow::printPages(QPainter *painter,
                             const QList<QStringList> &pages)
{
    int firstPage = printer.fromPage() - 1;
```

```
    if (firstPage >= pages.size())
        return;
    if (firstPage == -1)
        firstPage = 0;

    int lastPage = printer.toPage() - 1;
    if (lastPage == -1 || lastPage >= pages.size())
        lastPage = pages.size() - 1;

    int numPages = lastPage - firstPage + 1;
    for (int i = 0; i < printer.numCopies(); ++i) {
        for (int j = 0; j < numPages; ++j) {
            if (i != 0 || j != 0)
                printer.newPage();

            int index;
            if (printer.pageOrder() == QPrinter::FirstPageFirst) {
                index = firstPage + j;
            } else {
                index = lastPage - j;
            }
            printPage(painter, pages[index], index + 1);
        }
    }
}
```

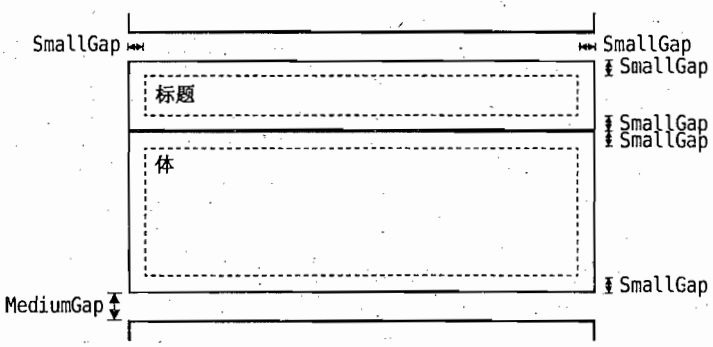


图 8.20 一种花的条目的布局

printPages()函数的作用就是使用 printPage(),并按照正确的顺序和正确的次数序号打印每一页。结果如图 8.21 所示。使用 QPrintDialog,用户可能要求几个副本,指定打印范围,或者倒序打印。提供这些选择是我们的责任——也可以用 QPrintDialog::setEnabledOptions()使它们失效。

首先确定打印的范围。QPrinter 的 fromPage()和 toPage()函数返回被用户选择的页码,或者没有被选中的就返回 0。减去 1,因为页列表是从 0 开始的。如果用户没有设置任何范围,则设置 firstPage 和 lastPage 为整个范围的值。

然后打印每一页。外面的循环按照用户所需要生成的副本的份数进行循环。绝大多数打印机驱动程序都支持多份副本,所以对于它们,QPrinter::numCopies()总是返回 1。如果打印机驱动程序不支持多份副本,numCopies()返回用户所需的副本数,并且应用程序会负责打印全部。[在前面 QImage 的例子中,为了简单忽略了 numCopies()。]

里面的 for 循环按页数循环。如果不是第 1 页,就调用 newPage()跳过旧的一页并且在一个新的空白页开始绘制。我们调用 printPage()绘制每一页。

```
void PrintWindow::printPage(QPainter *painter,
                            const QStringList &entries, int pageNumber)
{
    painter->save();
    painter->translate(0, LargeGap);
```

```
        foreach (QString entry, entries) {
            QStringList fields = entry.split(": ");
            QString title = fields[0];
            QString body = fields[1];
            printBox(painter, title, titleFont, Qt::lightGray);
            printBox(painter, body, bodyFont, Qt::white);
            painter->translate(0, MediumGap);
        }
        painter->restore();

        painter->setFont(footerFont);
        painter->drawText(painter->window(),
                          Qt::AlignHCenter | Qt::AlignBottom,
                          QString::number(pageNumber));
    }
```

printPage()函数遍历所有花的说明条目并且调用两个 printBox()打印它们:一个用于标题(花的名字),一个用于体(它的描述)。它也会在页的底部中央绘制页编号。页面的布局如图 8.22 所示。

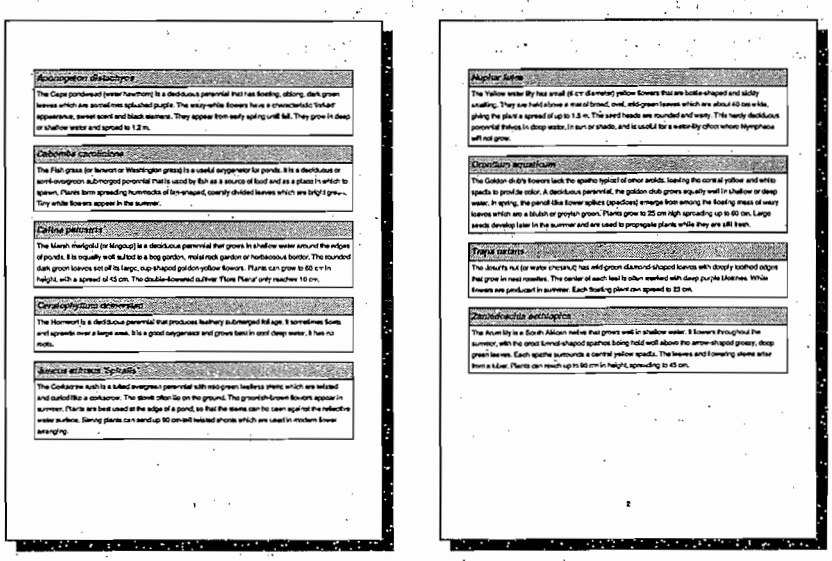


图 8.21　使用 QPainter 打印花的说明

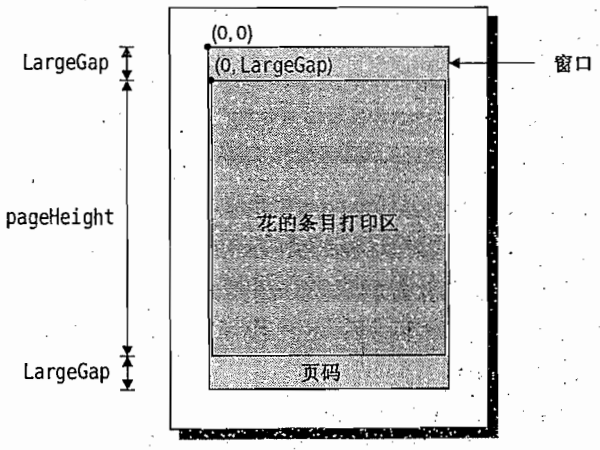


图 8.22　花的说明的页面布局

```
void PrintWindow::printBox(QPainter *painter, const QString &str,
                           const QFont &font, const QBrush &brush)
{
    painter->setFont(font);

    int boxWidth = painter->window().width();
    int textWidth = boxWidth - 2 * SmallGap;
    int maxHeight = painter->window().height();

    QRect textRect = painter->boundingRect(SmallGap, SmallGap,
                                           textWidth, maxHeight,
                                           Qt::TextWordWrap, str);
    int boxHeight = textRect.height() + 2 * SmallGap;

    painter->setPen(QPen(Qt::black, 2.0, Qt::SolidLine));
    painter->setBrush(brush);
    painter->drawRect(0, 0, boxWidth, boxHeight);
    painter->drawText(textRect, Qt::TextWordWrap, str);
    painter->translate(0, boxHeight);
}
```

printBox()函数绘制了一个框的边缘,然后绘制框里的文本。

至此,就完成了对二维图形打印的介绍。第 20 章将介绍三维图形。

第9章 拖 放

拖放是在一个应用程序内或者多个应用程序之间传递信息的一种直观的现代操作方式。除了为剪贴板提供支持外，通常它还提供数据移动和复制的功能。

在这一章中，将首先介绍如何为 Qt 应用程序添加支持拖放的功能以及如何处理自定义格式。然后，将介绍如何复用拖放代码来实现对剪贴板的支持。之所以能够复用这些代码，是因为拖放与剪贴板的功能机理均是以 QMimeData 类为基础的，而 QMimeData 是一个可以提供不同格式数据的类。

9.1 使拖放生效

拖放操作包括两个截然不同的动作：拖动和放下。Qt 窗口部件可以作为拖动点(darg site)、放下点(drop site)或者同时作为拖动点和放下点。

本节的第一个实例介绍了如何让一个 Qt 应用程序接受由另一个应用程序执行的一个拖动操作。该 Qt 应用程序是一个以 QTextEdit 作为中央窗口部件的主窗口程序。当用户从桌面上或从文件资源管理器中拖动一个文本文件并且在这个应用程序上放下时，该应用程序就会将文本文件载入到 QTextEdit 中。

以下是这个实例中 MainWindow 类的定义：

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

protected:
    void dragEnterEvent(QDragEnterEvent *event);
    void dropEvent(QDropEvent *event);

private:
    bool readFile(const QString &fileName);
    QTextEdit *textEdit;
};
```

MainWindow 类重新实现了来自 QWidget 的 dragEnterEvent()和 dropEvent()函数。由于该例子的主要目的是想说明拖放操作，所以一个主窗口类应该具有的很多功能在这里都被省略了。

```
MainWindow::MainWindow()
{
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);

    textEdit->setAcceptDrops(false);
    setAcceptDrops(true);

    setWindowTitle(tr("Text Editor"));
}
```

在构造函数中，创建了一个 QTextEdit 并且把它设置为中央窗口部件。默认情况下，QTextEdit 可以接受来自其他应用程序文本的拖动，并且如果用户在它上面放下一个文件，它将会把这个文

件的名称插入到文本中。由于拖放事件是从子窗口部件传递给父窗口部件的,所以通过禁用QTextEdit上的放下操作以及启用主窗口上的放下操作,就可以在整个MainWindow窗口中获得放下事件。

```
void MainWindow::dragEnterEvent(QDragEnterEvent *event)
{
    if (event->mimeData()->hasFormat("text/uri-list"))
        event->acceptProposedAction();
}
```

当用户把一个对象拖动到这个窗口部件上时,就会调用dragEnterEvent()。如果对这个事件调用acceptProposedAction(),就表明用户可以在这个窗口部件上拖放对象。默认情况下,窗口部件是不接受拖动的。Qt会自动改变光标来向用户说明这个窗口部件是不是有效的放下点。

这里,我们希望用户拖动的只能是文件,而非其他类型的东西。为了实现这一点,我们可以检查拖动的MIME类型。MIME类型中的text/uri-list用于存储一系列的统一资源标识符(Universal Resource Identifier,URI),它们可以是文件名、统一资源定位器(Uniform Resource Locator,URL,如HTTP或者FTP路径),或者其他全局资源标识符。标准的MIME类型是由国际因特网地址分配委员会(Internet Assigned Numbers Authority,IANA)定义的,它们由类型、子类型信息以及分隔两者的斜线组成。MIME类通常由剪贴板和拖放系统使用,以识别不同类型的数据。可以从下面的网站得到正式的MIME类型列表:http://www.iana.org/assignments/media-types/。

```
void MainWindow::dropEvent(QDropEvent *event)
{
    QList<QUrl> urls = event->mimeData()->urls();
    if (urls.isEmpty())
        return;

    QString fileName = urls.first().toLocalFile();
    if (fileName.isEmpty())
        return;

    if (readFile(fileName))
        setWindowTitle(tr("%1 - %2").arg(fileName)
                                    .arg(tr("Drag File")));
}
```

当用户在窗口部件上放下一个对象时,就会调用dropEvent()。我们调用函数QMimeData::urls()来获得QUrl列表。通常,用户一次只拖动一个文件,但是通过拖动一个选择区域来同时拖动多个文件也是可能的。如果要拖放的URL不止一个,或者要拖放的URL不是一个本地文件名,则会立即返回到原调用处。

QWidget也提供dragMoveEvent()和dragLeaveEvent()函数,但是在绝大多数应用程序中并不需要重新实现它们。

第2个实例描绘的是如何初始化一个拖动并且接受一个放下。我们将创建一个支持拖放的QListWidget子类,并且把它作为如图9.1所示的Project Chooser应用程序的一个组件。

Project Chooser应用程序为用户提供了两个由姓名组成的列表框。两个列表框分别代表一个项目。用户可以在两个项目列表框之间拖放这些姓名,将一个项目中的成员移动到另一个项目中。

拖放代码都放在QListWidget的子类中。以下是这个类的定义:

```
class ProjectListWidget : public QListWidget
{
    Q_OBJECT

public:
    ProjectListWidget(QWidget *parent = 0);
```

```
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dragMoveEvent(QDragMoveEvent *event);
    void dropEvent(QDropEvent *event);

private:
    void performDrag();

    QPoint startPos;
};
```

ProjectListWidget类重新实现了在QWidget中定义的5个事件处理器。

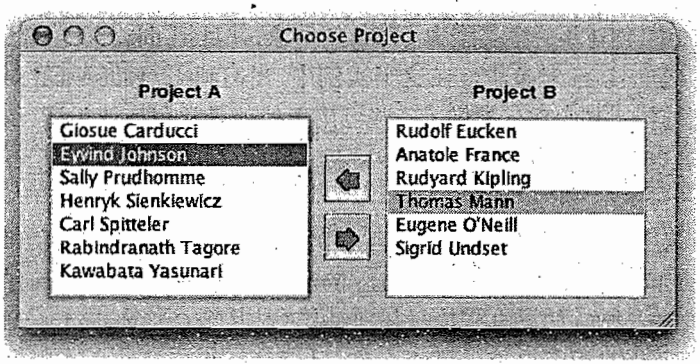


图9.1 Project Chooser应用程序

```
ProjectListWidget::ProjectListWidget(QWidget *parent)
    : QListWidget(parent)
{
    setAcceptDrops(true);
}
```

在构造函数中,我们使列表框上的放下生效。

```
void ProjectListWidget::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        startPos = event->pos();
    QListWidget::mousePressEvent(event);
}
```

当用户按下鼠标左键,就把鼠标位置保存到startPos私有变量中。我们调用QListWidget中mousePressEvent()的实现,以确保QListWidget可以像平常一样有机会处理鼠标按下的事件。

```
void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            performDrag();
    }
    QListWidget::mouseMoveEvent(event);
}
```

当用户按住鼠标左键并移动鼠标光标时,就认为这是一个拖动的开始。我们计算当前鼠标位置和原来鼠标左键按下的点之间的距离——这个“曼哈顿长度”(Manhattan Length)其实是从坐标原点到该矢量长度快速计算的近似值。如果这个距离大于或等于QApplication推荐的拖动起始距离值(通常是4个像素),那么就调用私有函数performDrag()以启动拖动操作。这可以避免用户因为手握鼠标抖动而产生拖动。

```
void ProjectListWidget::performDrag()
{
    QListWidgetItem *item = currentItem();
    if (item) {
        QMimeData *mimeData = new QMimeData;
        mimeData->setText(item->text());

        QDrag *drag = new QDrag(this);
        drag->setMimeData(mimeData);
        drag->setPixmap(QPixmap(":/images/person.png"));
        if (drag->exec(Qt::MoveAction) == Qt::MoveAction)
            delete item;
    }
}
```

在 performDrag()中,创建了一个类型为 QDrag 的对象,并且把 this 作为它的父对象。这个 QDrag 对象将数据存储在 QMimeData 对象中。在这个实例中,我们利用 QMimeData::setText()提供了作为 text/plain 字符串的数据。QMimeData 提供了一些可用于处理最常用拖放类型(诸如图像、URL、颜色,等等)的函数,同时也可以处理任意由 QByteArrays 表示的 MIME 类型。QDrag::setPixmap()调用则可以在拖放发生时使图标随光标移动。

QDrag::exec()调用启动并执行拖动操作,直到用户放下或取消此次拖动操作才会停止。它把所有支持的"拖放动作"(如 Qt::CopyAction,Qt::MoveAction 和 Qt::LinkAction)的组合作为其参数,并且返回被执行的拖放动作(如果没有执行任何动作,则返回 Qt::IgnoreAction)。至于执行的是哪个动作,取决于放下发生时源窗口部件是否允许、目标是否支持及按下了哪些组合键。在 exec()调用后,Qt 拥有拖动对象的所有权并且可以在不需要它的时候删除它。

```
void ProjectListWidget::dragEnterEvent(QDragEnterEvent *event)
{
    ProjectListWidget *source =
            qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

ProjectListWidget 窗口部件不仅能发起拖动,还可以接收同一个应用程序中来自另外一个 ProjectListWidget 部件的拖动。如果窗口部件是同一个应用程序的一部分,QDragEnterEvent::source()返回一个启动这个拖动的窗口部件的指针;否则,返回一个空指针值。我们使用qobject_cast <T> (),以确保这个拖动来自 ProjectListWidget。如果一切无误,就告诉 Qt 预备将该动作认为是一个移动动作。

```
void ProjectListWidget::dragMoveEvent(QDragMoveEvent *event)
{
    ProjectListWidget *source =
            qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

DragMoveEvent()中的代码与 dragEnterEvent()中编写的代码基本相同。这样是必要的,因为需要重写 QListWidget 的函数实现(实际上是 QAbstractItemView 的函数实现)。

```
void ProjectListWidget::dropEvent(QDropEvent *event)
{
    ProjectListWidget *source =
            qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
```

```
            addItem(event->mimeData()->text());
            event->setDropAction(Qt::MoveAction);
            event->accept();
        }
    }
```

在 dropEvent()中,我们使用 QMimeData::text()重新找回拖动的文本并随文本创建一个拖动项。还需要将事件作为"移动动作"来接受,从而告诉源窗口部件现在可以删除原来的拖动项了。

拖放是在应用程序之间传递数据的有力机制。但是在某些情况下,有可能在执行拖放时并未使用 Qt 的拖放工具。如果只是想在一个应用程序的窗口部件中移动数据,通常只要重新实现 mousePressEvent()和 mouseReleaseEvent()函数就可以了。

9.2 支持自定义拖动类型

目前为止所讲过的实例中,都是依靠 QMimeData 来支持通用的 MIME 类型。因此,我们调用 QMimeData::setText()来创建一个文本拖动,并且使用 QMimeData:urls()来重新找回 text/uri-list 拖动的内容。如果想拖动纯文本、超文本、图像、URL 甚至颜色,则可以只使用 QMimeData 类而不必采用其他的手段。但如果想拖动的是自定义数据,则必须选择如下三种方式之一:

1. 使用 QMimeData::setData(),可以提供任意数据作为 QByteArray 的内容,并且在随后利用 QMimeData::data()提取这些数据。
2. 可以通过子类化 QMimeData 并且重新实现 formats()和 retrieveData()来处理自定义数据类型。
3. 对于在简单应用程序中的拖放操作,可以子类化 QMimeData 并且利用我们所需要的任意数据结构来存储数据。

第 1 种方法虽不涉及任何的子类化,但它也有一些缺点:即使这个拖动没有被最后接收,仍需将数据结构转换为 QByteArray。另外,如果想提供几个 MIME 类型与应用程序在更广的范围内进行良好的交互,则必须对每一个 MIME 类型的数据进行存储。如果数据较大,这将不必要地减缓应用程序的运行速度。第 2 种和第 3 种方法则可以避免或者使这种问题的影响最小化,它们不仅将控制权完全交给用户,而且还可以一起使用。

为了说明这些方法是如何运作的,我们将给出一个如何给 QTableWidget 添加拖放功能的例子。拖动将支持以下的 MIME 类型:text/plain、text/html 和 text/csv。使用第 1 种方法,按如下的代码开始一个拖动:

```
void MyTableWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            performDrag();
    }
    QTableWidget::mouseMoveEvent(event);
}

void MyTableWidget::performDrag()
{
    QString plainText = selectionAsPlainText();
    if (plainText.isEmpty())
        return;

    QMimeData *mimeData = new QMimeData;
```

```
    mimeData->setText(plainText);
    mimeData->setHtml(toHtml(plainText));
    mimeData->setData("text/csv", toCsv(plainText).toUtf8());

    QDrag *drag = new QDrag(this);
    drag->setMimeData(mimeData);
    if (drag->exec(Qt::CopyAction | Qt::MoveAction) == Qt::MoveAction)
        deleteSelection();
}
```

在 mouseMoveEvent()中调用 performDrag()私有函数,可以拖拽出一个矩形选择框。我们使用 setText()和 setHtml()设置 text/plain 和 text/html 的 MIME 类型,同时使用 setData()设置 text/csv 类型,这些都可以采用任意的 MIME 类型和 QByteArray。selectionAsString()的代码或多或少地与第 4 章中介绍的 Spreadsheet::copy()函数的代码相似。

```
QString MyTableWidget::toCsv(const QString &plainText)
{
    QString result = plainText;
    result.replace("\\", "\\\\");
    result.replace("\"", "\\\"");
    result.replace("\t", "\", \"");
    result.replace("\n", "\"\n\"");
    result.prepend("\"");
    result.append("\"");

    return result;
}

QString MyTableWidget::toHtml(const QString &plainText)
{
    QString result = Qt::escape(plainText);
    result.replace("\t", "<td>");
    result.replace("\n", "\n<tr><td>");
    result.prepend("<table>\n<tr><td>");
    result.append("\n</table>");
    return result;
}
```

函数 toCsv()和 toHtml()将一个由制表符和换行符构成的字符串转换为一个 CSV 字符串(以逗号分隔的值,Comma-Separated Value)或者一个 HTML 字符串。例如,数据:

```
Red     Green   Blue
Cyan    Yellow  Magenta
```

可以转换为

```
"Red", "Green", "Blue"
"Cyan", "Yellow", "Magenta"
```

或者转换为

```
<table>
<tr><td>Red<td>Green<td>Blue
<tr><td>Cyan<td>Yellow<td>Magenta
</table>
```

利用 QString::replace(),可以用最简单的方式执行这个转换。另外,为了转义 HTML 中的特殊字符,还可以使用 Qt::escape()。

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
    if (event->mimeData()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeData()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
```

```
    } else if (event->mimeData()->hasFormat("text/plain")) {
        QString plainText = event->mimeData()->text();
        ...
        event->acceptProposedAction();
    }
}
```

尽管提供了三种不同格式的数据,但是在 dropEvent()事件中只接受其中的两种格式。如果用户从一个 QTableWidget 向某一 HTML 编辑器拖动单元格,我们将会希望这些单元格被转换成一个 HTML 表格。但是如果用户从任意的 HTML 向某一 QTableWidget 拖动,我们就不希望接受它。

为了完成这个例子,还需要在 MyTableWidget 构造函数中调用 setAcceptDrops(true)和 setSelectionMode(ContiguousSelection)。

我们将重做这个实例,但是这次将子类化 QMimeData 以延迟或者避免 QTableWidgetItem 与 QByteArray 之间(潜在耗时的)转换。下面是对子类的定义:

```
class TableMimeData : public QMimeData
{
    Q_OBJECT

public:
    TableMimeData(const QTableWidget *tableWidget,
                  const QTableWidgetSelectionRange &range);

    const QTableWidget *tableWidget() const { return myTableWidget; }
    QTableWidgetSelectionRange range() const { return myRange; }
    QStringList formats() const;

protected:
    QVariant retrieveData(const QString &format,
                          QVariant::Type preferredType) const;

private:
    static QString toHtml(const QString &plainText);
    static QString toCsv(const QString &plainText);

    QString text(int row, int column) const;
    QString rangeAsPlainText() const;

    const QTableWidget *myTableWidget;
    QTableWidgetSelectionRange myRange;
    QStringList myFormats;
};
```

我们存储一个 QTableWidgetSelectionRange 以取代存取实际的数据,QTableWidgetSelectionRange 详细说明了正在拖动的是哪些单元格,并且会始终保持一个指向 QTableWidget 的指针。来自 QMimeData 中的 formats()和 retrieveData()函数则会被重新实现。

```
TableMimeData::TableMimeData(const QTableWidget *tableWidget,
                             const QTableWidgetSelectionRange &range)
{
    myTableWidget = tableWidget;
    myRange = range;
    myFormats << "text/csv" << "text/html" << "text/plain";
}
```

在构造函数中,按上面代码段中的方法初始化私有变量。

```
QStringList TableMimeData::formats() const
{
    return myFormats;
}
```

formats()函数返回一个由 MIME 数据对象提供的 MIME 类型列表。各种格式之间的排列顺序

通常是无关联的,但是把"最好的"格式放在最前面是一种不错的习惯。支持多种格式的应用程序有时将使用第一个与其匹配的格式。

```
QVariant TableMimeData::retrieveData(const QString &format,
                                     QVariant::Type preferredType) const
{
    if (format == "text/plain") {
        return rangeAsPlainText();
    } else if (format == "text/csv") {
        return toCsv(rangeAsPlainText());
    } else if (format == "text/html") {
        return toHtml(rangeAsPlainText());
    } else {
        return QMimeData::retrieveData(format, preferredType);
    }
}
```

retrieveData()函数为给定的 MIME 类型返回一个数据作为其 QVariant。format 参数的值通常是由 formats()返回的字符串之一,但是我们并不能这样假设,因为并不是所有的应用程序都对照 formats()来检查 MIME 类型。由 QMimeData 提供的获取函数(getter function),诸如 text()、html()、urls()、imageData()、colorData()以及 data(),都可以根据 retrieveData()而重新实现。

preferredType 参数提示我们在 QVariant 中应该插入哪种类型。此处,如果有必要的话,我们将忽略 preferredType 参数而任由 QMimeData 将返回值转换为想要的类型。

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
    const TableMimeData *tableData =
            qobject_cast<const TableMimeData *>(event->mimeData());

    if (tableData) {
        const QTableWidget *otherTable = tableData->tableWidget();
        QTableWidgetSelectionRange otherRange = tableData->range();
        ...
        event->acceptProposedAction();
    } else if (event->mimeData()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeData()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
    } else if (event->mimeData()->hasFormat("text/plain")) {
        QString plainText = event->mimeData()->text();
        ...
        event->acceptProposedAction();
    }
    QTableWidget::mouseMoveEvent(event);
}
```

dropEvent()函数与本节早先曾提到的那个相似,但是这次首先检查是否可以成功地将 QMimeData 对象强制转换为 TableMimeData 来优化这个函数。如果 qobject_cast <T> () 有效,则意味着在同一个应用程序中,拖动是由 MyTableWidget 发起的,这样就可以直接读取表格数据而不必经过 QMimeData 的应用程序编程接口。如果强制转换失败,就以标准规定的方式提取数据。

在这个例子中,我们利用 UTF-8 格式编码 CSV 文本。如果想确信使用了正确的编码方式,则可以利用 text/plain 的 MIME 类型的 charset 参数来指定一个明确的编码方式。下面是几个明确指定编码的例子:

```
text/plain;charset=US-ASCII
text/plain;charset=ISO-8859-1
text/plain;charset=Shift_JIS
text/plain;charset=UTF-8
```

9.3 剪贴板处理技术

多数应用程序都通过某一种或者几种方式来使用 Qt 的内置剪贴板处理技术。例如，QTextEdit 类除了提供快捷键支持外，还提供 cut()、copy()和 paste()槽，所以几乎不需要其他额外代码。

当编写自己的类时，可以通过 QApplication::clipboard()访问剪贴板，它会返回一个指向应用程序 QClipboard 对象的指针。处理系统剪贴板是很容易的：调用 setText()、setImage()或者 setPixmap()把数据放到剪贴板中，并且调用 text()、image()和 pixmap()来重新获得数据即可。已经在第 4 章中的 Spreadsheet 应用程序中看到了一些剪贴板应用的例子。

对于某些应用程序，仅使用内置的功能可能是不够的。例如，也许我们想提供的数据不是文本或者图像类型；或者也许我们想按几种不同的格式提供数据以最大限度地与其他应用程序进行协同工作。这个问题和之前在拖放中所遇到的问题一样，并且解决方案也很相似：必须子类化 QMimeData，并且重新实现一些虚函数。

如果应用程序通过一个自定义 QMimeData 子类支持拖放，则可以只复用 QMimeData 子类并且利用 setMimeData()函数把它放到剪贴板中。可以对剪贴板调用 mimeData()来获得数据。

在 X11 上，通常可以点击一个三键鼠标的中键来粘贴选项。这个操作可以通过使用一个单独的“选定”剪贴板完成。如果希望窗口部件像支持标准剪贴板一样也支持这种“选项”剪贴板，那么必须将 QClipboard::Selection 作为一个额外的参数传递给各种剪贴板的调用。例如，以下是如何在文本编辑器中重新执行 mouseReleaseEvent()以支持鼠标中键粘贴功能的代码：

```
void MyTextEditor::mouseReleaseEvent(QMouseEvent *event)
{
    QClipboard *clipboard = QApplication::clipboard();
    if (event->button() == Qt::MidButton
            && clipboard->supportsSelection()) {
        QString text = clipboard->text(QClipboard::Selection);
        pasteText(text);
    }
}
```

在 X11 上，supportsSelection()函数会返回 true 值；而在其他平台上，它返回 false。

如果想在剪贴板中的内容发生改变时就立即得到通报，那么可以通过建立 QClipboard::dataChanged()信号和自定义槽的连接来实现。

第10章 项视图类

许多应用程序允许用户搜索、查看和编辑属于某个数据集中的一些个别项。这些数据可能保存在文件中、数据库中或者网络服务器上。处理像这样的数据集的标准方式是使用 Qt 的项视图类(item view class)。

在早期的 Qt 版本中,项视图窗口部件总是由一个数据集的所有内容组装而成的。用户在这个窗口部件的数据上进行所有的查询和编辑操作,并且在某些情况下,对数据的改变还会被重新回写到数据源中。尽管这种方式很容易理解和使用,但是在使用非常大的数据集的时候,这种方式就不能很好地工作了,并且如果我们想在两个或者更多的窗口部件中显示同一个数据集的时候,这种方式就不能很好地适应。

Smalltalk 语言普及了一种非常灵活的对于大数据集的可视化方法:模型-视图-控制器(Model-View-Controller,MVC)。在 MVC 方法中,模型(model)代表数据集,它对需要查看数据的获取以及任何存储的改变负责。每种类型的数据集都有自己的模型,但不管底层的数据集是什么样子,模型提供给视图(view)的 API 都是相同的。视图代表的是面向用户的那些数据。在同一时间,任何大数据集只有有限的部分是可见的,所以这个有限的部分就是视图所请求的那部分数据。控制器(controller)是用户和视图之间的媒介,它把用户的操作转换为浏览或者编辑数据的请求,这部分数据是根据需要由视图传送给模型的数据。

根据 MVC 方法的启示,Qt 提供了一种模型/视图架构,如图 10.1 所示。在 Qt 中,这个模型和它在经典的 MVC 中的行为是完全相同的。Qt 使用的不是控制器,而是使用了一种稍微有些不同的抽象:委托(delegate)。委托用于对项的如何显示和如何编辑提供精细控制。Qt 对每种类型的视图都提供了默认的委托。对于绝大多数应用程序,这已经足够了,所以我们通常不需要注意它。

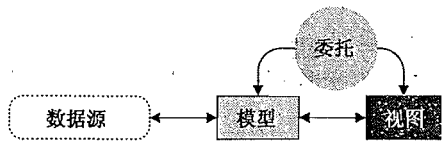


图 10.1 Qt 的模型/视图体系结构

利用 Qt 的模型/视图架构,我们可以只从模型中获取实际在视图中显示所需要的数据。这样在处理非常大的数据集的时候,可以更加快速,而不至于降低性能。通过把一个模型注册到两个或者多个视图,就可以让用户使用不同的方式查看数据以及和数据交互。Qt 对于多个视图会自动地保持同步,从而使对一个视图的改变会影响到全部视图,如图 10.2 所示。模型/视图架构的另外一个好处是:如果决定改变底层数据集的存储方式,则只需要修改模型,而视图仍将能够继续正常工作。

在很多情况下,只需要把一小部分的项呈现给用户。在这些常见的例子中,可以使用 Qt 提供的那些方便的项视图类(QListWidget、QTableWidget 和 QTreeWidget),并且可以把它们和项直接组装起来。这些类和一些以前的 Qt 版本中提供的项视图类很相似。它们把数据存储在"项"中(例如,QTableWidget 就包含了一些 QTableWidgetItem)。实际上,在这些方便的类的内部,使用了自定义的模型,这样就可以让这些项在视图中变为可见的了。

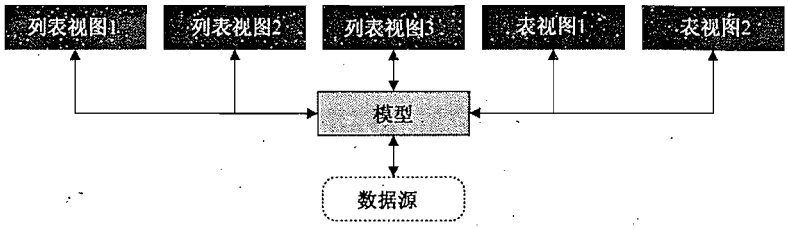


图 10.2　适用于多种视图的模型

对于大数据集,数据复制通常不是一种好方法。这时可以使用 Qt 的视图(QListView、QTableView 和 QTreeView),把它们和一个数据模型连接,而这个数据模型既可以是自定义模型,也可以是 Qt 的预定义模型。例如,如果数据集被保存在一个数据库中,就可以把 QTableView 和 QSqlTableModel 组合在一起使用。

10.1　使用项视图的简便类

使用 Qt 的项视图中那些方便的子类通常要比定义一个自定义模型简单得多,并且如果我们不需要由区分模型和视图所带来的好处时,这种方法也是比较合适的。在第 4 章中,当我们继承 QTableWidget 和 QTableWidgetItem 来实现电子制表软件的功能的时候,就已经使用了这一技术。

这一节将说明如何使用这些方便的项视图子类来显示项。第一个例子演示了一个只读的 QListWidget(如图 10.3 所示),第二个例子演示了一个可以编辑的 QTableWidget(如图 10.4 所示),第三个例子则演示了一个只读的 QTreeWidget(如图 10.5 所示)。

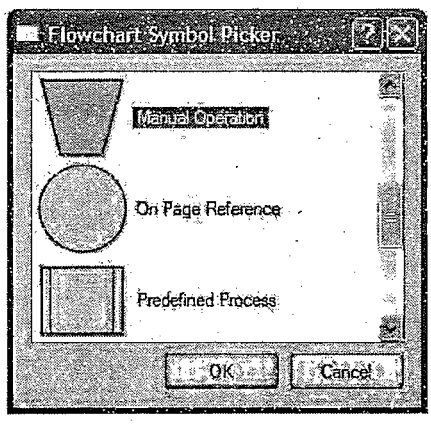


图 10.3　流程图符号选择器应用程序

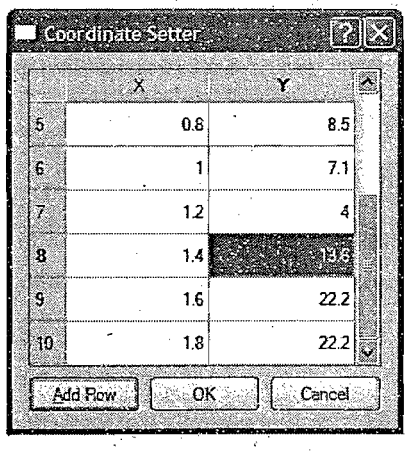


图 10.4　坐标设置器应用程序

先从一个允许用户在列表中选择流程图符号的简单对话框开始。每个项都由一个图标、一段文本和一个唯一的 ID 组成。

先从这个对话框的头文件中截取的代码开始:

```
class FlowChartSymbolPicker : public QDialog
{
    Q_OBJECT

public:
    FlowChartSymbolPicker(const QMap<int, QString> &symbolMap,
                          QWidget *parent = 0);
```

```
    int selectedId() const { return id; }
    void done(int result);
    ...
};
```

当构造这个对话框时,我们必须给它传递一个 QMap <int, QString> ,在它运行之后,可以通过调用 selectedId()获得一个选中的 ID(或者如果用户没有选中任何一项,就返回 -1)。

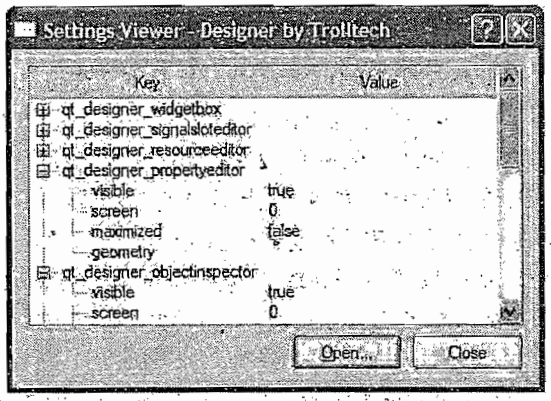


图 10.5 设置查看器应用程序

```
FlowChartSymbolPicker::FlowChartSymbolPicker(
        const QMap<int, QString> &symbolMap, QWidget *parent)
    : QDialog(parent)
{
    id = -1;

    listWidget = new QListWidget;
    listWidget->setIconSize(QSize(60, 60));

    QMapIterator<int, QString> i(symbolMap);
    while (i.hasNext()) {
        i.next();

        QListWidgetItem *item = new QListWidgetItem(i.value(),
                                                    listWidget);
        item->setIcon(iconForSymbol(i.value()));
        item->setData(Qt::UserRole, i.key());
    }
    ...
}
```

我们把 id(最后被选择的 ID)初始化为 -1。接下来构造一个方便的项视图窗口部件 QListWidget。我们遍历这个流程图符号映射中的每一个项,并且为了显示每一个项而创建一个 QListWidgetItem。QListWidgetItem 构造函数都会在父对象 QListWidget 之后获得一个代表将要显示的文本的 QString。

然后,设置项的图标并且调用 setData()函数把任意 ID 保存到 QListWidgetItem 中。iconForSymbol()私有函数为每一个给定的符号名称返回一个 QIcon。

QListWidgetItem 有几个角色(role),每一个角色都有一个关联的 QVariant。最常用的角色有 Qt::DisplayRole、Qt::EditRole 和 Qt::IconRole,并且这些角色都有方便的设置和获取函数[setText()和 setIcon()],另外还有其他几个角色。通过指定一个大于等于 Qt::UserRole 的值,就可以定义自定义的角色。在我们的例子中,使用 Qt::UserRole 存储每一个项的 ID。

构造函数中省略的部分包括创建按钮、对这些窗口部件进行摆放以及设置这个窗口的标题栏。

```
void FlowChartSymbolPicker::done(int result)
{
    id = -1;
    if (result == QDialog::Accepted) {
        QListWidgetItem *item = listWidget->currentItem();
        if (item)
            id = item->data(Qt::UserRole).toInt();
    }
    QDialog::done(result);
}
```

这个 done()函数是由 QDialog 重新实现的。当用户单击 OK 或者 Cancel 按钮时,就会调用它。如果用户单击 OK,就获得相应的项并且使用 data()函数提取 ID。如果对项的文本感兴趣,则可以通过调用 item -> data(Qt::DisplayRole).toString()或者更为方便的 item -> text()来获取文本。

默认情况下,QListWidget 是只读的。如果想让用户编辑这些项,则可以使用 QAbstractItemView::setEditTriggers()设置这个视图的编辑触发器(edit trigger)。例如,QAbstractItemView::AnyKeyPressed 这个设置值的意思是:用户只要一开始输入就进入项的编辑状态。类似地,也可以提供一个 Edit(编辑)按钮(还可以提供 Add 按钮和 Delete 按钮),同时使用信号-槽连接,这样就可以使用程序来控制编辑操作了。

现在已经看到了如何使用方便的项视图类来查看和选择数据,接下来将会看一个可以编辑数据的例子。再次使用一个对话框,它显示的是用户可以编辑的一对(x,y)坐标。

正如前面例子中所做的那样,这里也从构造函数开始,并着重讲解有关项视图的代码。

```
CoordinateSetter::CoordinateSetter(QList<QPointF> *coords,
                                   QWidget *parent)
    : QDialog(parent)
{
    coordinates = coords;

    tableWidget = new QTableWidget(0, 2);
    tableWidget->setHorizontalHeaderLabels(
            QStringList() << tr("X") << tr("Y"));

    for (int row = 0; row < coordinates->count(); ++row) {
        QPointF point = coordinates->at(row);
        addRow();
        tableWidget->item(row, 0)->setText(QString::number(point.x()));
        tableWidget->item(row, 1)->setText(QString::number(point.y()));
    }
    ...
}
```

这个 QTableWidget 构造函数得到要在这个表格中显示的行和列的初始数字。在 QTableWidget 中的每一个项都使用一个 QTableWidgetItem 表示,包括水平方向和垂直方向的表头项。setHorizontalHeaderLabels()函数则设置每一个表窗口部件的水平表头项的文本为所传递的字符串列表中的相应文本。默认情况下,QTableWidget 会提供一个垂直表头,这个列的标签从 1 开始,这正是我们想要的,所以不需要手工设置垂直表头的标签。

一旦创建完列标签,就可以遍历传递进来的坐标数据。对于每一个(x,y)坐标对,都添加一个新行[使用私有函数 addRow()],同时在每一行的列中设置合适的文本。

默认情况下,QTableWidget 允许编辑。用户可以在遍历一个表的时候,按下 F2 或者任意简单的输入,来编辑这个表的任意单元格。用户在这个视图中所做的任何修改都会自动影响这些 QTableWidgetItem。为了防止编辑,可以调用 setEditTriggers(QAbstractItemView::NoEditTriggers)。

```
void CoordinateSetter::addRow()
{
```

```
    int row = tableWidget->rowCount();

    tableWidget->insertRow(row);

    QTableWidgetItem *item0 = new QTableWidgetItem;
    item0->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 0, item0);

    QTableWidgetItem *item1 = new QTableWidgetItem;
    item1->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 1, item1);

    tableWidget->setCurrentItem(item0);
}
```

当用户单击 Add Row 按钮时,就会调用这个 addRow()槽,这种方式在函数构造中也经常使用到。我们使用 QTableWidget::insertRow()插入一个新的行,然后创建两个 QTableWidgetItem()项,并利用 QTableWidget::insertRow()把它们添加到表中。除了该项外,QTableWidget::setItem()还需要一行及一列。最后,我们设置当前项,这样用户就可以开始编辑新的一行的第一项了。

```
void CoordinateSetter::done(int result)
{
    if (result == QDialog::Accepted) {
        coordinates->clear();
        for (int row = 0; row < tableWidget->rowCount(); ++row) {
            double x = tableWidget->item(row, 0)->text().toDouble();
            double y = tableWidget->item(row, 1)->text().toDouble();
            coordinates->append(QPointF(x, y));
        }
    }
    QDialog::done(result);
}
```

最后,当用户单击 OK 时,就清空传递给这个对话框的坐标,并且根据这个 QTableWidget 的所有项创建一个新的坐标集。

在关于 Qt 的简便项视图窗口部件的第三个也是最后一个例子中,我们将会查看一些应用程序的代码片段,它使用 QTreeWidget 显示 Qt 应用程序设置。QTreeWidget 默认是只读的。

以下是构造函数的一部分:

```
SettingsViewer::SettingsViewer(QWidget *parent)
    : QDialog(parent)
{
    organization = "Trolltech";
    application = "Designer";

    treeWidget = new QTreeWidget;
    treeWidget->setColumnCount(2);
    treeWidget->setHeaderLabels(
            QStringList() << tr("Key") << tr("Value"));
    treeWidget->header()->setResizeMode(0, QHeaderView::Stretch);
    treeWidget->header()->setResizeMode(1, QHeaderView::Stretch);
    ...
    setWindowTitle(tr("Settings Viewer"));
    readSettings();
}
```

为了访问应用程序的设置,必须使用组织名称和应用程序名称作为参数创建 QSettings 对象。我们设置了默认的名称(Trolltech 的 Designer),然后构造一个新的 QTreeWidget。树形窗口部件的头视图控制了树形列队的大小。我们设置两列的重定义模式为 Stretch。这就告诉了头视图总是确保列能够填充有用的空间。在这种模式下,用户或者程序都不能重新调整列的大小。在构造函数的最后,我们调用这个 readSettings()函数来构成树形窗口部件。

```
void SettingsViewer::readSettings()
{
    QSettings settings(organization, application);

    treeWidget->clear();
    addChildSettings(settings, 0, "");

    treeWidget->sortByColumn(0);
    treeWidget->setFocus();

    setWindowTitle(tr("Settings Viewer - %1 by %2")
                   .arg(application).arg(organization));
}
```

应用程序的设置会存储在一个由键和值组成的分层结构中。addChildSettings()私有函数需要一个设置对象、一个父对象的 QTreeWidgetItem 和当前"组"(group)。QSettings 群组相当于文件系统目录。addChildSettings()函数可以递归调用自己来遍历任意一个树状结构。readSettings()里是初始调用,并且传递一个作为父项的空指针,表示这是根。

```
void SettingsViewer::addChildSettings(QSettings &settings,
        QTreeWidgetItem *parent, const QString &group)
{
    if (!parent)
        parent = treeWidget->invisibleRootItem();
    QTreeWidgetItem *item;

    settings.beginGroup(group);

    foreach (QString key, settings.childKeys()) {
        item = new QTreeWidgetItem(parent);
        item->setText(0, key);
        item->setText(1, settings.value(key).toString());
    }
    foreach (QString group, settings.childGroups()) {
        item = new QTreeWidgetItem(parent);
        item->setText(0, group);
        addChildSettings(settings, item, group);
    }
    settings.endGroup();
}
```

addChildSettings()函数用于创建所有的 QTreeWidgetItem。它在设置树的当前层次中遍历所有键并且为每个键创建一个 QTableWidgetItem。如果空指针被作为 parent 项传递进来,就创建一个以 QTreeWidget::invisibleRootItem()为父对象的项,这样它就成为了顶层项。第一列设置为键的名称,第二列设置为相对应的值。

接下来,这个函数在当前层中遍历每一个组。对于每一个组,都会创建一个新的 QTreeWidgetItem,并且把它的第一列设置为这个组的名称。然后这个函数会把这个群组项作为父对象递归调用自己,这样就可以用这个组的子项来构成这个 QTreeWidget 了。

在这一节中所显示的项视图窗口部件允许我们使用和以前的 Qt 版本中类似的方式进行编程:读取要被设置到一个项视图窗口部件中的所有数据、使用项对象呈现数据元素并且回写数据源(如果这些项是可编辑的话)。在后面的几节中,我们将不再局限于这种简单的方式,并且会充分利用 Qt 模型/视图架构的所有好处。

10.2 使用预定义模型

Qt 提供了几种可以在视图类中使用的预定义模型,见下表。

QStringListModel	存储一个字符串列表
QStandardItemModel	存储任意的分层次的数据
QDirModel	封装本地文件系统
QSqlQueryModel	封装一个 SQL 结果集
QSqlTableModel	封装一个 SQL 表
QSqlRelationalTableModel	利用外键封装一个 SQL 表
QSortFilterProxyModel	排序和/或筛选另一个模型

这一节将看到如何使用 QStringListModel、QDirModel 和 QSortFilterProxyModel。SQL 模型将会在第 13 章中讲到。

我们从一个简单的对话框开始,用户可以使用它添加、删除和编辑一个 QStringList,其中每个字符串都代表一个团队领导。对话框如图 10.6 所示。

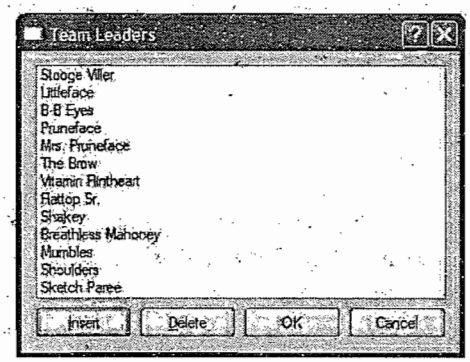


图 10.6 团队领导应用程序

以下是从构造函数中提取的部分相关代码:

```
TeamLeadersDialog::TeamLeadersDialog(const QStringList &leaders,
                                     QWidget *parent)
    : QDialog(parent)
{
    model = new QStringListModel(this);
    model->setStringList(leaders);

    listView = new QListView;
    listView->setModel(model);
    listView->setEditTriggers(QAbstractItemView::AnyKeyPressed
                              | QAbstractItemView::DoubleClicked);
    ...
}
```

我们从创建并且组装一个 QStringListModel 开始。接下来创建一个 QListView 并且把刚才创建的那个模型作为它的模型。还设置了一些编辑触发器以允许用户简单地通过开始输入或者双击进入编辑字符串的状态。默认情况下,QListView 中没有任何编辑触发器,这样就使这个视图只读。

```
void TeamLeadersDialog::insert()
{
    int row = listView->currentIndex().row();
    model->insertRows(row, 1);

    QModelIndex index = model->index(row);
    listView->setCurrentIndex(index);
    listView->edit(index);
}
```

当用户单击 Insert 按钮的时候，insert()槽就会得到调用。这个槽从获得列表视图的当前项的行数开始。模型中的每一个数据项都对应一个“模型索引”，它是由一个 QModelIndex 对象表示的。会在下一节中看到有关模型索引的细节，但是现在我们需要知道一个索引有三个主要组成部分：行、列和它所属的模型的指针。在一个一维的列表视图中，列总为 0。

一旦得到行数，就可以在那个位置插入一个新行。这个插入是在模型中完成的，并且模型会自动更新列表视图。然后我们设置刚刚插入的空白行为列表视图的当前索引。最后，设置列表视图在当前行进入编辑的状态，就好像用户已经按下一个键或者双击进入编辑状态一样。

```
void TeamLeadersDialog::del()
{
    model->removeRows(listView->currentIndex().row(), 1);
}
```

在构造函数中，Delete 按钮的 clicked()信号会连接到这个 del()槽。因为我们只是想删除当前行，所以可以使用当前索引位置调用 removeRows()，并且把要删除的行数设置为 1。就像刚才的插入操作一样，我们需要依赖于模型才能够相应地更新这个视图。

```
QStringList TeamLeadersDialog::leaders() const
{
    return model->stringList();
}
```

最后，当关闭这个对话框的时候，leaders()函数提供了一种读回这些被编辑的字符串的方式。

通过简单地改变 TeamLeadersDialog 的窗口标题，它就可以成为一个通用的字符串列表编辑对话框。我们通常所需要的另外一个通用对话框是向用户显示一个文件或者目录的列表。下一个实例(如图 10.7 所示)就使用了 QDirModel 类，它封装了计算机的文件系统并且可以显示(或者隐藏)不同的文件属性。可以为这个模型应用过滤器，这样就可以根据自己的需要显示不同类型的文件系统条目，并且使用不同的方式对这些条目进行排序。

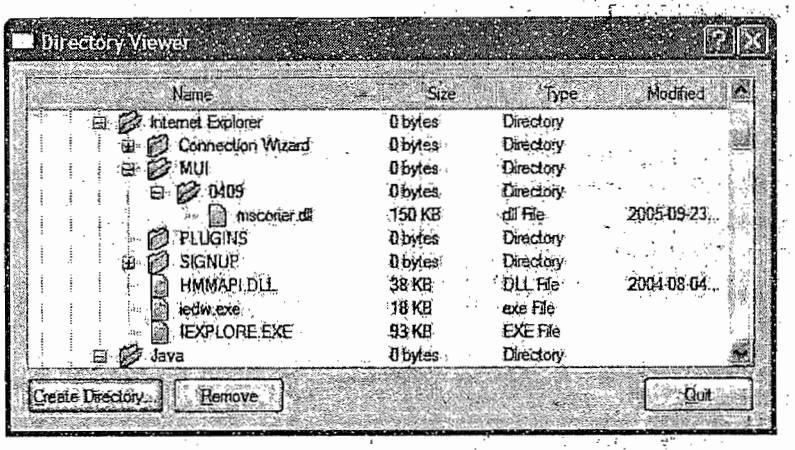


图 10.7　Directory Viewer 应用程序

我们将从 Directory Viewer 对话框的构造函数开始，看看是如何创建和设置这个模型和视图的。

```
DirectoryViewer::DirectoryViewer(QWidget *parent)
    : QDialog(parent)
{
    model = new QDirModel;
    model->setReadOnly(false);
    model->setSorting(QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);

    treeView = new QTreeView;
```

```
    treeView->setModel(model);
    treeView->header()->setStretchLastSection(true);
    treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
    treeView->header()->setSortIndicatorShown(true);
    treeView->header()->setClickable(true);

    QModelIndex index = model->index(QDir::currentPath());
    treeView->expand(index);
    treeView->scrollTo(index);
    treeView->resizeColumnToContents(0);
    ...
}
```

一旦模型构造完成,我们就让它可编辑并且设置不同的初始排序属性。然后创建用于显示这个模型的数据的 QTreeView。这个 QTreeView 的头可以用来提供用户控制的排序功能。通过让头可以点击,用户就能够按他们所点击的列进行排序。当这个树视图的头被设置完毕之后,就得到了当前目录的模型索引,然后通过调用 expand(),如果需要就打开它的父对象一直到根节点,并且调用 scrollTo()滚动到当前项,这样就确保它是可见的。然后我们确保第一列足够宽,可以显示它所有的条目,而不是使用省略号。

这里没有给出的构造函数中的部分代码中,把 Create Directory 和 Remove 按钮和实现这些操作的槽连接起来。我们不需要 Rename 按钮,这是因为用户可以通过按下 F2 键并键入字母来直接进行重命名。

```
void DirectoryViewer::createDirectory()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;

    QString dirName = QInputDialog::getText(this,
                                tr("Create Directory"),
                                tr("Directory name"));
    if (!dirName.isEmpty()) {
        if (!model->mkdir(index, dirName).isValid())
            QMessageBox::information(this, tr("Create Directory"),
                    tr("Failed to create the directory"));
    }
}
```

如果用户在输入对话框中输入的是一个目录名称,就试图会在当前目录下用这个名称创建一个子目录。QDirModel::mkdir()函数可以使用父目录的索引和新目录的名称,然后返回它所创建的目录的模型索引。如果操作失败,就返回一个无效的模型索引。

```
void DirectoryViewer::remove()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;

    bool ok;
    if (model->fileInfo(index).isDir()) {
        ok = model->rmdir(index);
    } else {
        ok = model->remove(index);
    }
    if (!ok)
        QMessageBox::information(this, tr("Remove"),
                tr("Failed to remove %1").arg(model->fileName(index)));
}
```

如果用户单击 Remove,就试图移除当前项所对应的文件或者目录。也可以使用 QDir 来完成这项操作,但是 QDirModel 提供了一些对 QModelIndex 起作用的方便函数。

本节的最后一个实例(如图 10.8 所示)显示了如何使用 QSortFilterProxyModel。和其他预定义模型不同,这个模型封装了一个已经存在的模型并且对在底层模型和视图之间的传递的数据进行操作。在我们的实例中,底层模型是一个由 Qt 所认识的颜色名称[通过调用 QColor::colorNames()得到]初始化的 QStringListModel。用户可以在 QLineEdit 中输入一个过滤器字符串并且使用组合框指定这个字符串被如何解释(作为正则表达式、通配符模式或者固定字符串)。



图 10.8　颜色名称应用程序

以下是 ColorNamesDialog 构造函数的一部分:

```
ColorNamesDialog::ColorNamesDialog(QWidget *parent)
    : QDialog(parent)
{
    sourceModel = new QStringListModel(this);
    sourceModel->setStringList(QColor::colorNames());

    proxyModel = new QSortFilterProxyModel(this);
    proxyModel->setSourceModel(sourceModel);
    proxyModel->setFilterKeyColumn(0);

    listView = new QListView;
    listView->setModel(proxyModel);
    ...
    syntaxComboBox = new QComboBox;
    syntaxComboBox->addItem(tr("Regular expression"), QRegExp::RegExp);
    syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
    syntaxComboBox->addItem(tr("Fixed string"), QRegExp::FixedString);
    ...
}
```

我们使用常规的方式创建并且组装这个 QStringListModel。接下来就是对 QSortFilterProxyModel 的构造。我们使用 setSourceModel()传递底层模型并且告诉这个代理过滤器被应用在初始模型的第 0 列。QComboBox::addItem()函数接受一个类型为 QVariant 的可选"数据"参数。我们使用它存储和每一个项文本对应的 QRegExp::PatternSyntax 值。

```
void ColorNamesDialog::reapplyFilter()
{
    QRegExp::PatternSyntax syntax =
            QRegExp::PatternSyntax(syntaxComboBox->itemData(
                    syntaxComboBox->currentIndex()).toInt());
    QRegExp regExp(filterLineEdit->text(), Qt::CaseInsensitive, syntax);
    proxyModel->setFilterRegExp(regExp);
}
```

只要用户改变过滤器字符串或者模式语法组合框,就会调用这个 reapplyFilter()槽。我们使用行编辑器中的文本创建一个 QRegExp,然后设置它的模式语法为保存在组合框中当前项中保存的值。当调用 setFilterRegExp()时,会激活新的过滤器并且会自动更新视图。

10.3　实现自定义模型

Qt 的预定义模型为数据的处理和查看提供了很好的方法。但是,有些数据源不能有效地和预定义模型一起工作,这时就需要创建自定义模型,以方便对底层数据源进行优化。

在介绍如何创建自定义模型之前,让我们先看看在 Qt 模型/视图架构中的一些重点概念。在模型中,每一个数据元素都有一个模型索引和一套属性(attribute),称为角色(role),这些角色可以

保存任意值。本章的前面部分中已看到过最常用的角色 Qt::DisplayRole 和 Qt::EditRole。其他角色都是用来补充说明数据的(例如,Qt::ToolTipRole、Qt::StatusTipRole 和 Qt::WhatsThisRole),还有一些是用来控制基本显示属性的(例如,Qt::FontRole、Qt::TextAlignmentRole、Qt::TextColorRole 和 Qt::BackgroundColorRole)。

对于列表模型,唯一和索引部分相关的就是行号,可以通过 QModelIndex::row()得到。对于表模型,与索引部分相关的就是行号和列号,分别可以通过 QModelIndex::row()和 QModelIndex::column()得到。对于列表模型和表模型,每一个项的父对象都是根,通常由一个无效的 QModelIndex 表示。本节的前两个实例就是如何实现自定义表模型。

树模型和表模型类似,但有如下不同之处:像表模型一样,最顶层项的父对象是根(一个无效的 QModelIndex),但是其他每一个项的父对象都是继承树中的其他一些项。这些父对象可以通过 QModelIndex::parent()得到。每一个项都有自己的角色数据,都有 0 个或多个子对象,每一个项都有属于自己的东西。因为项可以拥有其他项作为子对象,这样它就可以用来显示递归(树形的)的数据结构,本节的最后一个例子将会讲述这一部分。图 10.9 给出了不同模型的示意图。

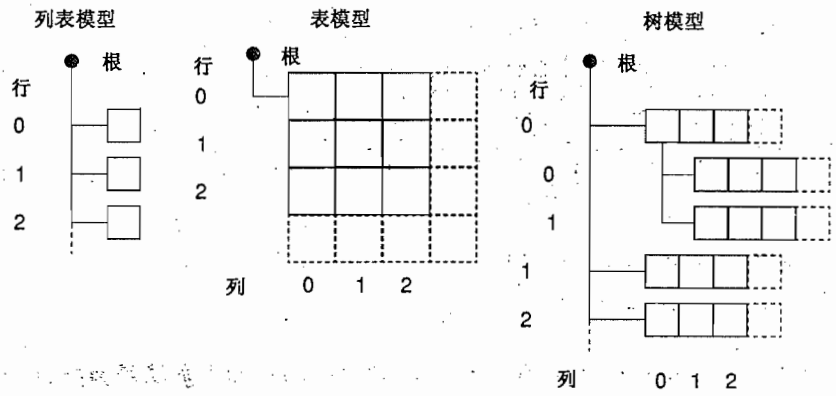


图 10.9 Qt 模型示意图

本节的第一个例子是一个显示各个货币之间汇率关系的只读表结构,如图 10.10 所示。

Currencies

	NOK	NZD	SEK	SGD	USD
NOK	1.0000	0.2254	1.1991	0.2592	0.1534
NZD	4.4363	1.0000	5.3195	1.1500	0.6804
SEK	0.8340	0.1880	1.0000	0.2162	0.1279
SGD	3.8578	0.8696	4.6258	1.0000	0.5917
USD	6.5200	1.4697	7.8180	1.6901	1.0000

图 10.10 汇率应用程序

本可以用一个简单的表模型实现这个例子,但是我们想使用一个自定义模型,它可以使我们得到数据的常用属性并且节省存储空间。如果要在一个表中存储 162 个货币单位之间的汇率,则需要存储 162 × 162 = 26 244 个值。利用本节的自定义 CurrencyModel 模型,只需要存储 162 个值(每一个货币和美元之间的汇率值)。

CurrencyModel 类将和一个标准的 QTableView 一同使用。CurrencyModel 由 QMap <QString, double>

组装而成，每一个键都是一个货币编码，并且每一个值都是以美元为单位的货币值。下面的这段代码显示的就是如何组装这个映射和如何使用这个模型：

```
QMap<QString, double> currencyMap;
currencyMap.insert("AUD", 1.3259);
currencyMap.insert("CHF", 1.2970);
...
currencyMap.insert("SGD", 1.6901);
currencyMap.insert("USD", 1.0000);

CurrencyModel currencyModel;
currencyModel.setCurrencyMap(currencyMap);

QTableView tableView;
tableView.setModel(&currencyModel);
tableView.setAlternatingRowColors(true);
```

现在查看一下这个模型的实现。从头文件开始：

```
class CurrencyModel : public QAbstractTableModel
{
public:
    CurrencyModel(QObject *parent = 0);

    void setCurrencyMap(const QMap<QString, double> &map);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;

private:
    QString currencyAt(int offset) const;

    QMap<QString, double> currencyMap;
};
```

我们选择 QAbstractTableModel 作为模型的基类，这是因为它和数据源最为接近。Qt 提供了几个模型基类，其中包括 QAbstractListModel、QAbstractTableModel 和 QAbstractItemModel，见图 10.11。QAbstractItemModel 类用于支持很多种模型，其中包括那些基于递归数据结构的模型，而 QAbstractListModel 和 QAbstractTableModel 两个类主要用于提供一维和二维数据集。

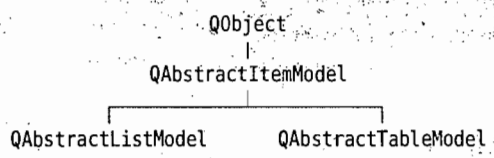


图 10.11 抽象模型类的继承树

对于只读的表模型，必须重新实现三个函数：rowCount()、columnCount()和 data()。在这种情况下，还需要重新实现 headerData()，并且还提供了一个初始化数据的函数[setCurrencyMap()]。

```
CurrencyModel::CurrencyModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

在构造函数中，除了要把 parent 参数传递给基类之外，不需要再做其他任何事情。

```
int CurrencyModel::rowCount(const QModelIndex & /* parent */) const
{
    return currencyMap.count();
```

```
}

int CurrencyModel::columnCount(const QModelIndex & /* parent */) const
{
    return currencyMap.count();
}
```

对于这种表模型,行号和列号就是这个汇率映射中货币的种类。parent 参数对于表模型没有任何意义。之所以在此处保留这个参数,是因为 rowCount()和 columnCount()都是从更加通用的 QAbstractItemModel 这个基类中继承的,在这个类中支持层次结构。

```
QVariant CurrencyModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        QString rowCurrency = currencyAt(index.row());
        QString columnCurrency = currencyAt(index.column());

        if (currencyMap.value(rowCurrency) == 0.0)
            return "####";

        double amount = currencyMap.value(columnCurrency)
                        / currencyMap.value(rowCurrency);

        return QString("%1").arg(amount, 0, 'f', 4);
    }
    return QVariant();
}
```

data()函数返回一个项的任意角色的值,这个项被指定为 QModelIndex。对于表模型,QModelIndex 中有意义的部分是它的行号和列号,可以通过调用 row()和 column()得到它们。

如果角色是 Qt::TextAlignmentRol,就返回一个与数字相匹配的对齐方式;如果角色是 Qt::DisplayRole,就查找每一种货币对应的值并且计算出兑换汇率。

可以返回 double 类型的计算结果,但是那样就无法控制显示的精度(除非使用的是一个自定义的委托)。所以,我们返回字符串类型的值,这个字符串已经按照我们的想法进行了格式化。

```
QVariant CurrencyModel::headerData(int section,
                                   Qt::Orientation /* orientation */,
                                   int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    return currencyAt(section);
}
```

当视图组装水平表头和垂直表头时,就会调用 headerData()函数。section 参数是指行号或者列号(这取决于实际方向)。因为这里的行和列都是相同的货币代码,所以不需要考虑方向,只需简单地根据给定的序号返回相应货币的代码即可。

```
void CurrencyModel::setCurrencyMap(const QMap<QString, double> &map)
{
    currencyMap = map;
    reset();
}
```

调用者可以使用 setCurrencyMap()函数改变货币映射。QAbstractItemModel::reset()调用告诉任何一个使用这个模型的视图,它们所有的数据都无效了,这样就会强制它们为可见的项刷新数据。

```
QString CurrencyModel::currencyAt(int offset) const
{
    return (currencyMap.begin() + offset).key();
}
```

currencyAt()函数返回在货币映射中给定位移的键(即货币代码)。我们使用 STL 风格的迭代器查找这个条目并且对它调用 key()。

正如现在所看到的,依赖于底层数据的实际情况创建只读模型并不困难,而且具有良好设计的模型还可以节约内存,提升速度。下一个实例,即图 10.12 所示的城市应用程序也是基于表的,但是这一次所有的数据都是由用户输入的。

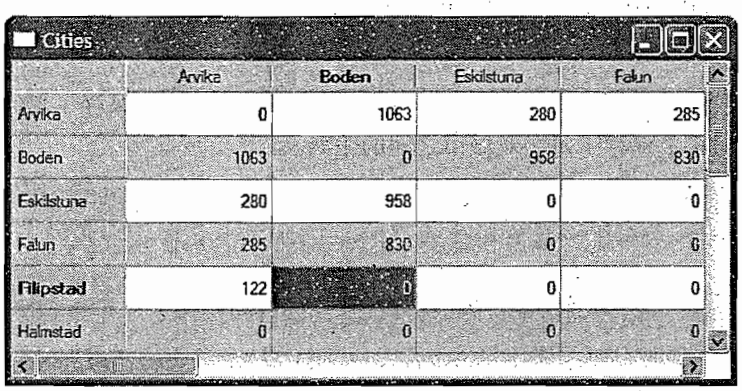


图 10.12　城市应用程序

这个应用程序用于存储任意两个城市之间的距离的值。像前一个实例一样,也可以简单地使用 QTableWidget,并且为每一个城市对存储一个项。但是,一个自定义的模型将会更加有效率,因为无论从任意城市 A 到另外任意一个不同的城市 B,或者是从 B 到 A,两者之间的距离都是相同的,所以这些项将会是按主对角线而对称。

为了查看自定义模型和简单表之间的区别,我们假设有三个城市:A、B 和 C。如果为每一个组合存储一个值,就需要存储 9 个值。而一个仔细设计过的模型只需要三个项:(A,B)、(A,C)和(B,C)。

下面的代码建立并使用这一模型:

```
QStringList cities;
cities << "Arvika" << "Boden" << "Eskilstuna" << "Falun"
       << "Filipstad" << "Halmstad" << "Helsingborg" << "Karlstad"
       << "Kiruna" << "Kramfors" << "Motala" << "Sandviken"
       << "Skara" << "Stockholm" << "Sundsvall" << "Trelleborg";

CityModel cityModel;
cityModel.setCities(cities);

QTableView tableView;
tableView.setModel(&cityModel);
tableView.setAlternatingRowColors(true);
```

和前一个实例一样,我们必须重新实现一些相同的函数。另外,为了使模型可以被编辑,还必须重新实现 setData()和 flags()。下面是类的定义:

```
class CityModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    CityModel(QObject *parent = 0);
```

```
    void setCities(const QStringList &cityNames);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    bool setData(const QModelIndex &index, const QVariant &value,
                 int role);
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;

private:
    int offsetOf(int row, int column) const;

    QStringList cities;
    QVector<int> distances;
};
```

对于这个模型,使用了两个数据结构:类型为 QStringList 的 cities 保存了城市的名称,类型为 QVector <int> 的 distances 保存了每一对城市之间的距离。

```
CityModel::CityModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

构造函数除了把 parent 参数传递给基类外,就没有其他任何的操作了。

```
int CityModel::rowCount(const QModelIndex & /* parent */) const
{
    return cities.count();
}

int CityModel::columnCount(const QModelIndex & /* parent */) const
{
    return cities.count();
}
```

因为是一个正方形的城市列表,所以行总数和列总数就是列表中的城市总数。

```
QVariant CityModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        if (index.row() == index.column())
            return 0;
        int offset = offsetOf(index.row(), index.column());
        return distances[offset];
    }
    return QVariant();
}
```

这个 data()函数和我们在 CurrencyModel 中所做的类似。如果行和列相同,它返回 0,因为它对应当两个城市相同的时候的情况。否则,它在 distances 矢量中查找给定的行和列所对应的条目,并且返回这个特定城市对的距离。

```
QVariant CityModel::headerData(int section,
                               Qt::Orientation /* orientation */,
                               int role) const
{
    if (role == Qt::DisplayRole)
        return cities[section];
    return QVariant();
}
```

headerData()函数非常简单,因为表是正方形的,它的水平表头和垂直表头一样。我们可以简单地返回在 cities 字符串列表中给定偏移量的城市名称。

```
bool CityModel::setData(const QModelIndex &index,
                        const QVariant &value, int role)
{
    if (index.isValid() && index.row() != index.column()
            && role == Qt::EditRole) {
        int offset = offsetOf(index.row(), index.column());
        distances[offset] = value.toInt();

        QModelIndex transposedIndex = createIndex(index.column(),
                                                  index.row());
        emit dataChanged(index, index);
        emit dataChanged(transposedIndex, transposedIndex);
        return true;
    }
    return false;
}
```

当用户编辑一个项的时候,就会调用 setData()函数。提供的模型索引必须有效,两个城市必须不同,并且当修改的数据元素是 Qt::EditRole 时,这个函数会把用户输入的值存储到 distances 矢量中。

createIndex()函数用于产生一个模型索引。我们需要使用它获得在主对角线另外一侧和当前正在被设置的项所对应项的模型索引,因为这两个项必须显示相同的数据。createIndex()函数中的参数顺序是行号在列号之前。这里调换这两个参数,这样就可以获得由 index 指定的项所对应项的模型索引。

我们用被改变的项的模型索引作为参数发射 dataChanged()信号。这个信号使用两个参数的原因是:它可以用来表示对一个矩形范围影响的改变,而不仅仅是一个项,所以这里传递的是被改变区域的左上角和右下角两个项的索引。我们还对另外一个对应的项发射 dataChanged()信号,以确保视图刷新它。最后,返回 true 或者 false 表明编辑操作是否成功。

```
Qt::ItemFlags CityModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags flags = QAbstractItemModel::flags(index);
    if (index.row() != index.column())
        flags |= Qt::ItemIsEditable;
    return flags;
}
```

模型会使用 flags()函数得到该如何对一个项进行相关的操作(例如,是否可以编辑)。在 QAbstractTableModel 中的默认实现是返回 Qt::ItemIsSelectable | Qt::ItemIsEnabled。对于所有不在主对角线上的项(它们总是 0),都会添加这个 Qt::ItemIsEditable 标记。

```
void CityModel::setCities(const QStringList &cityNames)
{
    cities = cityNames;
    distances.resize(cities.count() * (cities.count() - 1) / 2);
    distances.fill(0);
    reset();
}
```

如果给定一个新的城市列表,就设置私有的 QStringList 为新的列表,重新定义 distances 矢量的大小,清空所有的值,并且调用 QAbstractItemModel::reset()通知所有视图,它们的可见项必须被重新获取。

```
int CityModel::offsetOf(int row, int column) const
{
    if (row < column)
```

```
        qSwap(row, column);
    return (row * (row - 1) / 2) + column;
}
```

offsetOf()私有函数用于计算给定的城市对在 distances 矢量中的索引。例如,如果有城市 A、B、C 和 D,并且用户更新第 3 行第 1 列,也就是从 B 到 D,则偏移量应该是 $3 \times (3-1)/2+1=4$。如果用户要更新第 1 行第 3 列,也就是从 D 到 B,则要归功于 qSwap()的调用,它会执行相同的计算,并且也将返回同样的偏移量。图 10.13 阐明了城市、距离及其相应表模型之间的关系。

城市

A	B	C	D

距离

A↔B	A↔C	A↔D	B↔C	B↔D	C↔D

表模型

	A	B	C	D
A	0	A↔B	A↔C	A↔D
B	A↔B	0	B↔C	B↔D
C	A↔C	B↔C	0	C↔D
D	A↔D	B↔D	C↔D	0

图 10.13 城市和距离的数据结构以及表模型

本节的最后一个例子是显示给定布尔表达式的解析树的模型。布尔表达式可以是一个简单的包含文字和数字的标识符,如"bravo",也可以是一个利用"&&"、"||"或"!"等算子连接简单表达式的复合表达式,还可以是括号表达式。例如"a||(b && !c)"就是一个布尔表达式。

如图 10.14 所示的布尔解析器应用程序由 4 个类组成:

- BooleanWindow 是一个让用户输入布尔表达式并且显示相应解析树的窗口。
- BooleanParser 从一个布尔表达式生成一个解析树。
- BooleanModel:是一个封装解析树的树模型。
- Node 代表解析树中的一个节点。

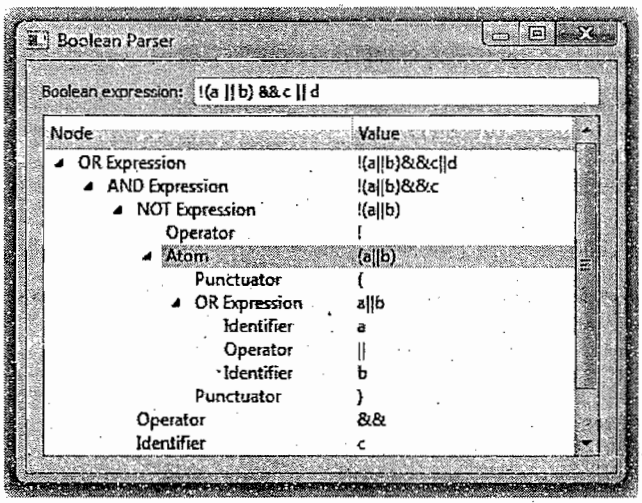


图 10.14 布尔解析器应用程序

让我们从 Node 类开始:

```
class Node
{
public:
    enum Type { Root, OrExpression, AndExpression, NotExpression, Atom,
                Identifier, Operator, Punctuator };
```

```
    Node(Type type, const QString &str = "");
    ~Node();

    Type type;
    QString str;
    Node *parent;
    QList<Node *> children;
};
```

每一个节点都有一个类型、一个字符串(可以为空)、一个父对象(可以为 null)和一个子节点的列表(可以为空)。

```
Node::Node(Type type, const QString &str)
{
    this->type = type;
    this->str = str;
    parent = 0;
}
```

构造函数只是简单地初始化这个节点的类型和字符串,并将父对象设置为 null(即没有父对象)。因为所有数据都是公有的,使用 Node 的代码可以直接操作类型、字符串、父对象和子节点。

```
Node::~Node()
{
    qDeleteAll(children);
}
```

qDeleteAll()函数遍历一个容器的所有指针并且对每一个指针都调用 delete。但是它并不把这些指针设置为 null,所以在一个析构函数外面使用这个函数时,通常在它之后需要调用 clear(),以清空这个容器中保存的所有指针。

现在我们已经定义了数据项(每一个都表示为一个 Node),已经为创建一个模型做好了准备:

```
class BooleanModel : public QAbstractItemModel
{
public:
    BooleanModel(QObject *parent = 0);
    ~BooleanModel();

    void setRootNode(Node *node);

    QModelIndex index(int row, int column,
                      const QModelIndex &parent) const;
    QModelIndex parent(const QModelIndex &child) const;

    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;

private:
    Node *nodeFromIndex(const QModelIndex &index) const;

    Node *rootNode;
};
```

这一次使用 QAbstractItemModel 作为基类,而不是它的方便的子类 QAbstractTableModel,这是因为我们想创建一个分层模型。必须重新实现的最基本的函数仍然相同,除此之外还需要实现 index()和 parent()。为了设置模型的数据,还有一个 setRootNode()函数,调用这个函数的时候,必须使用解析树的根节点作为参数。

```
BooleanModel::BooleanModel(QObject *parent)
    : QAbstractItemModel(parent)
{
    rootNode = 0;
}
```

在模型的构造函数中,只需要把根节点设置为一个安全的 null 值并且把 parent 传递给基类。

```
BooleanModel::~BooleanModel()
{
    delete rootNode;
}
```

在析构函数中,我们删除根节点。如果根节点有子对象,每一个子对象也都会被删除,并且依次递归往下删除,这是由 Node 的析构函数完成的。

```
void BooleanModel::setRootNode(Node *node)
{
    delete rootNode;
    rootNode = node;
    reset();
}
```

当设置新的根节点设置时,我们从删除任何之前的根节点(和它所有的子对象)开始。然后设置新的根节点并且调用 reset()通知所有视图,它们必须为任何一个可见的项重新获取数据。

```
QModelIndex BooleanModel::index(int row, int column,
                                const QModelIndex &parent) const
{
    if (!rootNode || row < 0 || column < 0)
        return QModelIndex();
    Node *parentNode = nodeFromIndex(parent);
    Node *childNode = parentNode->children.value(row);
    if (!childNode)
        return QModelIndex();
    return createIndex(row, column, childNode);
}
```

index()函数是由 QAbstractItemModel 重新实现的。只要模型或者视图需要为一个特定的子项(或者如果 parent 是一个无效的 QModelIndex 时为顶级项)创建一个 QModelIndex 的时候,这个函数就会被调用。对于表模型和列表模型,不需要重新实现这个函数,因为 QAbstractListModel 和 QAbstractTableModel 默认的实现已经足够了。

在 index()实现中,如果没有设置解析树,就返回一个无效的 QModelIndex。否则,就根据给定的行和列以及一个 Node * 为被请求的子对象创建一个 QModelIndex。对于层次模型,已知一个项相对于它的父对象的行和列并不能够唯一确定它,还必须知道它的父对象是谁。为了解决这个问题,可以在 QModelIndex 中存储一个指向内部节点的指针。除了行号和列号之外,QModelIndex 还为我们提供了存储一个 void * 或者 int 的选择。

可以通过父节点的 children 列表获得子节点的 Node * 。我们使用 nodeFromIndex()私有函数从 parent 模型索引中提取父节点:

```
Node *BooleanModel::nodeFromIndex(const QModelIndex &index) const
{
    if (index.isValid()) {
        return static_cast<Node *>(index.internalPointer());
    } else {
        return rootNode;
    }
}
```

nodeFromIndex()函数把给定索引的 void * 强制转换成为 Node *，或者如果这个索引是无效的，就返回根节点，因为在模型中，一个无效的模型索引用来表示根。

```
int BooleanModel::rowCount(const QModelIndex &parent) const
{
    if (parent.column() > 0)
        return 0;
    Node *parentNode = nodeFromIndex(parent);
    if (!parentNode)
        return 0;
    return parentNode->children.count();
}
```

一个给定项的行总数就是它有多少个子对象。

```
int BooleanModel::columnCount(const QModelIndex & /* parent */) const
{
    return 2;
}
```

列总数被固定为 2。第一列用来保存节点类型，第二列用来保存节点值。

```
QModelIndex BooleanModel::parent(const QModelIndex &child) const
{
    Node *node = nodeFromIndex(child);
    if (!node)
        return QModelIndex();
    Node *parentNode = node->parent;
    if (!parentNode)
        return QModelIndex();
    Node *grandparentNode = parentNode->parent;
    if (!grandparentNode)
        return QModelIndex();

    int row = grandparentNode->children.indexOf(parentNode);
    return createIndex(row, 0, parentNode);
}
```

从一个子节点获得父节点的 QModelIndex，要比查找一个父节点的子节点多做一些工作。我们可以很容易地通过使用 nodeFromIndex()得到一个节点，然后通过使用这个 Node 的父指针获得父节点，但是为了获得行号(父节点在它的层次中的位置)，还需要继续向上到祖父节点并且查找这个父节点在它的父节点(也就是这个子节点的祖父节点)中的索引位置。

```
QVariant BooleanModel::data(const QModelIndex &index, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();

    Node *node = nodeFromIndex(index);
    if (!node)
        return QVariant();

    if (index.column() == 0) {
        switch (node->type) {
        case Node::Root:
            return tr("Root");
        case Node::OrExpression:
            return tr("OR Expression");
        case Node::AndExpression:
            return tr("AND Expression");
        case Node::NotExpression:
            return tr("NOT Expression");
        case Node::Atom:
            return tr("Atom");
        case Node::Identifier:
            return tr("Identifier");
        case Node::Operator:
```

```
            return tr("Operator");
        case Node::Punctuator:
            return tr("Punctuator");
        default:
            return tr("Unknown");
        }
    } else if (index.column() == 1) {
        return node->str;
    }
    return QVariant();
}
```

在 data()中,可以获得请求的项的 Node *,并且我们使用它访问底层的数据。如果调用者想得到 Qt::DisplayRole 之外的其他任何角色的值,或者如果不能从给定的模型索引中获得一个 Node,就返回一个无效的 QVariant。如果列为 0,就返回这个节点类型的名称;如果列为 1,就返回这个节点的值(也就是它的字符串)。

```
QVariant BooleanModel::headerData(int section,
                                  Qt::Orientation orientation,
                                  int role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
        if (section == 0) {
            return tr("Node");
        } else if (section == 1) {
            return tr("Value");
        }
    }
    return QVariant();
}
```

在 headerData()的重新实现中,我们返回合适的水平表头标签。在用于可视化层次模型的 QTreeView 类中,没有垂直表头,所以可以忽略它了。

现在已经完成了 Node 和 BooleanModel 两个类,让我们看看当用户在行编辑器中改变文本的时候根节点是如何创建的:

```
void BooleanWindow::booleanExpressionChanged(const QString &expr)
{
    BooleanParser parser;
    Node *rootNode = parser.parse(expr);
    booleanModel->setRootNode(rootNode);
}
```

用户改变这个应用程序的行编辑器的文本时,会调用主窗口的 booleanExpression()槽。在这个槽中,用户的文本被解析并且解析器返回这个解析树的根节点的指针。

这里没有讲解 BooleanParser 类,是因为它和 GUI 或者模型/视图编程无关。随书的示例中给出了这个实例的完整源代码。

当实现像 BooleanModel 之类的树模型时,很容易出错,从而导致 QTreeView 出现奇怪的行为。为了帮助并解决自定义数据模型中的问题,Trolltech 实验室提供了一个 ModelTest 类。这个类对模型执行一系列测试,以发现常见的错误。如果想使用 ModelTest,可以从网址 http://labs.trolltech.com/page/Projects/Itemview/Modeltest 下载,并按照 README 文件中给定的说明进行操作。

在这一节中,我们已经看到了如何创建三种不同的自定义模型。很多模型要比这里显示的更为简单,只要在项和模型索引中做好一一对应即可。Qt 中还提供了更多的模型/视图的实例以及大量的文档资料,具体请查阅附录或者帮助文档。

10.4 实现自定义委托

委托(delegate)用来渲染和编辑视图中不同的项。在大多数情况下,视图中默认的委托已经足够了。如果想更好地控制有关项的显示,通常可以通过使用自定义模型很简单地实现我们所想要的:在 data()重新实现中,我们可以处理 Qt::FontRole、Qt::TextAlignmentRole、Qt::TextColorRole 和 Qt::BackgroundColorRole,并且它们会被默认的委托使用。例如,在之前所显示的城市和汇率例子中,我们已经为了得到向右对齐的数字的效果处理了 Qt::TextAlignmentRole。

如果想得到更多的控制,则可以创建自己的委托类并且把它设置在我们想要使用它的视图中。下面显示的音轨编辑器对话框就使用了一个自定义委托,它显示了音轨的标题和持续时间。模型中保存的数据是非常简单的 QString(标题)和 int(秒),但是持续时间将会被分隔成分钟和秒两部分,并且会通过 QTimeEdit 来让它变得可以编辑。

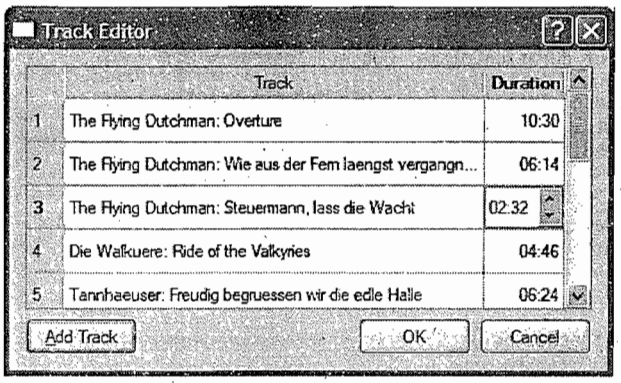


图 10.15 音轨编辑器对话框

音轨编辑器对话框使用了一个 QTableWidget,这是一个可以在 QTableWidgetItem 上操作的方便的项视图子类。提供的数据是一个 Track 列表:

```
class Track
{
public:
    Track(const QString &title = "", int duration = 0);

    QString title;
    int duration;
};
```

下面显示的是 TaskEditor 构造函数中的一部分,它用来创建和组装这个表窗口部件:

```
TrackEditor::TrackEditor(QList<Track> *tracks, QWidget *parent)
    : QDialog(parent)
{
    this->tracks = tracks;

    tableWidget = new QTableWidget(tracks->count(), 2);
    tableWidget->setItemDelegate(new TrackDelegate(1));
    tableWidget->setHorizontalHeaderLabels(
            QStringList() << tr("Track") << tr("Duration"));

    for (int row = 0; row < tracks->count(); ++row) {
        Track track = tracks->at(row);

        QTableWidgetItem *item0 = new QTableWidgetItem(track.title);
        tableWidget->setItem(row, 0, item0);
```

```
        QTableWidgetItem *item1
            = new QTableWidgetItem(QString::number(track.duration));
        item1->setTextAlignment(Qt::AlignRight);
        tableWidget->setItem(row, 1, item1);
    }
    ...
}
```

这个构造函数创建了一个表窗口部件,并没有简单地使用默认的委托,而是设置了自定义的TrackDelegate,传递保存时间数据的列作为参数。我们由设置列的表头开始,然后遍历数据,利用每一个音轨的名称和持续时间组装各行。

构造函数中的其余部分以及 TrackEditor 对话框的其余部分中没有什么特别的地方,所以现在我们将直接查看 TrackDelegate 是如何处理音轨数据的显示和编辑的。

```
class TrackDelegate : public QItemDelegate
{
    Q_OBJECT

public:
    TrackDelegate(int durationColumn, QObject *parent = 0);

    void paint(QPainter *painter, const QStyleOptionViewItem &option,
               const QModelIndex &index) const;
    QWidget *createEditor(QWidget *parent,
                          const QStyleOptionViewItem &option,
                          const QModelIndex &index) const;
    void setEditorData(QWidget *editor, const QModelIndex &index) const;
    void setModelData(QWidget *editor, QAbstractItemModel *model,
                      const QModelIndex &index) const;

private slots:
    void commitAndCloseEditor();

private:
    int durationColumn;
};
```

我们使用 QItemDelegate 作为基类,所以可以从默认的委托实现中获益。如果想从头开始做,则可以使用 QAbstractItemDelegate 作为基类[①]。为了提供一个可以编辑数据的委托,必须实现 createEditor()、setEditorData()和 setModelData()。还要实现 paint(),它用于改变持续时间这一列的显示。

```
TrackDelegate::TrackDelegate(int durationColumn, QObject *parent)
    : QItemDelegate(parent)
{
    this->durationColumn = durationColumn;
}
```

构造函数中的 durationColumn 参数告诉这个委托,哪一列保存的是音轨的持续时间。

```
void TrackDelegate::paint(QPainter *painter,
                          const QStyleOptionViewItem &option,
                          const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QString text = QString("%1:%2")
                       .arg(secs / 60, 2, 10, QChar('0'))
                       .arg(secs % 60, 2, 10, QChar('0'));

        QStyleOptionViewItem myOption = option;
```

① Qt 4.4 有望引入一个 QStyledItemDelegate 类并把它作为默认的委托。与 QItemDelegate 不同,QStyledItemDelegate 将依赖于当前风格以绘制出它的项。

```
        myOption.displayAlignment = Qt::AlignRight | Qt::AlignVCenter;

        drawDisplay(painter, myOption, myOption.rect, text);
        drawFocus(painter, myOption, myOption.rect);
    } else{
        QItemDelegate::paint(painter, option, index);
    }
}
```

因为我们想使用“分钟:秒”的格式显示持续时间,所以需要重新实现这个 paint()函数。两个 arg()调用中的参数分别为:要显示为字符串的整数、需要的字符串有多少个字符、整数的基数(10代表十进制)和填充的字符。

为了右对齐文本,我们复制当前的风格选项并且覆盖默认的对齐方式。然后调用 QItemDelegate::drawDisplay()绘制这个文本,然后跟着的是 QItemDelegate::drawFocus(),如果当前项具有焦点,它就绘制一个焦点矩形,否则什么也不做。使用 drawDisplay()非常方便,特别是当使用了自定义的风格选项的时候。还可以使用绘图器直接绘制。

```
QWidget *TrackDelegate::createEditor(QWidget *parent,
        const QStyleOptionViewItem &option,
        const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = new QTimeEdit(parent);
        timeEdit->setDisplayFormat("mm:ss");
        connect(timeEdit, SIGNAL(editingFinished()),
                this, SLOT(commitAndCloseEditor()));
        return timeEdit;
    } else {
        return QItemDelegate::createEditor(parent, option, index);
    }
}
```

我们只想控制音轨持续时间的编辑,所以把音轨名称的编辑留给了默认的委托。这是通过检查要求委托提供一个编辑器的是哪一列实现这一点的。如果是持续时间所在的列,就创建一个 QTimeEdit,设置正确的显示格式,并且把它的 editingFinished()信号和 commitAndCloseEditor()槽连接起来。对于其他任何一列,都把有关编辑的处理传递给默认的委托。

```
void TrackDelegate::commitAndCloseEditor()
{
    QTimeEdit *editor = qobject_cast<QTimeEdit *>(sender());
    emit commitData(editor);
    emit closeEditor(editor);
}
```

如果用户按下 Enter 键或者把焦点移动到这个 QTimeEdit 之外(但不是按下 Esc 键),editingFinished()信号就会被发射并且就会调用 commitAndCloseEditor()槽。这个槽会发射 commitData()信号,通知视图用被编辑的数据替换已经存在的数据。它还发射 closeEditor()信号,通知视图已经不再需要这个编辑器了,这时模型将会把它删除。编辑器可以使用 QObject::sender()获得,这个函数返回发射了触发这个槽的信号的对象。如果用户取消编辑(按下 Esc 键),视图将会简单地删除编辑器。

```
void TrackDelegate::setEditorData(QWidget *editor,
                                  const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        timeEdit->setTime(QTime(0, secs / 60, secs % 60));
    } else {
```

```
        QItemDelegate::setEditorData(editor, index);
    }
}
```

当用户初始化编辑的时候,视图会调用 createEditor()创建一个编辑器,然后利用这个项的当前数据调用 setEditorData()来初始化编辑器。如果编辑器是用于持续时间列的,就按秒提取这个音轨的持续时间,并且设置 QTimeEdit 的时间为相应的分钟数和秒数;否则,就让默认的委托处理这个初始化。

```
void TrackDelegate::setModelData(QWidget *editor,
                                 QAbstractItemModel *model,
                                 const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        QTime time = timeEdit->time();
        int secs = (time.minute() * 60) + time.second();
        model->setData(index, secs);
    } else {
        QItemDelegate::setModelData(editor, model, index);
    }
}
```

如果用户完成了编辑(例如,在这个编辑器窗口部件外面按下鼠标左键,或者按下了 Enter 键或 Esc 键),而不是取消编辑,模型就必须使用编辑器的数据进行更新。如果持续时间被编辑了,就从 QTimeEdit 中提取出分钟数和秒数,并且设置数据为相应的秒数。

我们完全有可能创建一个可以很好地控制模型中任何一个项的编辑和显示的自定义委托,尽管在这种情况下不是必需的。我们已经选择控制特定的列,但是因为 QModelIndex 会被传递给重新实现的所有 QItemDelegate 的函数,所以可以按照行、列、矩形区域、父对象或者它们中的任意组合进行控制,如果需要还可以控制每一个单独的项。

在这一章中,我们已经为 Qt 的模型/视图搭建了一个总体的框架,演示了如何使用视图方便的子类、如何使用 Qt 预定义的模型和如何创建自定义模型和自定义委托。但是模型/视图架构是如此的丰富,以至于没有足够的空间讲解它所能做到的一切。例如,可以创建一个自定义视图,并它把它的项呈现为列表、表和树以外的形式。在 Qt 的 examples/itemviews/chart 目录下的 Chart 实例讲解了这种用法,它显示了如何自定义一个把模型数据显示为饼状图形式的视图。

另外,也可以使用多个视图查看同一个模型,而不需要繁琐的工作。通过其中一个视图的任何编辑操作都会自动并且立即影响其他视图。在查看大型数据,当用户希望能够同时查看逻辑上截然不同的几个部分的数据的时候,这种功能非常有用。这种架构也支持选择:当两个或者多个视图正在使用同一个模型时,每一个视图都可以被设置为拥有自己独立的选择,或者可以在不同的视图中共享选择。

Qt 的在线文档对于项视图的形成以及它的类实现提供了很详尽的介绍。所有与项视图相关的类的列表,可以参考 http://doc.trolltech.com/4.3/model-view.html。有关其他信息以及 Qt 中所包含的相关实例的链接,请参考 http://doc.trolltech.com/4.3/model-view-programming.html。

第11章 容 器 类

容器类通常是用于在内存中存储给定类型的许多项的模板类。C++已经提供了很多容器,作为标准模板库(STL)的一部分,它们都包含在标准的C++库中。

Qt提供了属于自己的容器类,所以在编写Qt程序时,既可以使用Qt容器也可以使用STL容器。Qt容器的主要优点是它们在所有的平台上在运行时都表现得一致,并且它们都是隐含共享的。隐含共享(implicit sharing),或者称为"写时复制",是一个能够把整个容器作为不需要太多运行成本的值来传递的最优化过程。Qt容器的另一个主要特征就是易于使用的迭代器类,这是从Java中得到的灵感,它们可以利用QDataStream变成数据流,而且它们通常可以使可执行文件中的代码量比相应的STL类中的要少。最后,在Qt/Embedded Linux支持的一些硬件平台上,通常只能使用Qt容器。

Qt既提供了诸如QVector <T>、QLinkedList <T>和QList <T>等的连续容器,也提供了诸如QMap <K,T>和QHash <K,T>等的关联容器。从概念上分析,顾名思义,连续容器连续地存储项,而关联容器则存储键值对。

Qt还提供了在任意容器上执行相关操作的通用算法。例如,qSort()算法对一个连续容器进行排序,qBinaryFind()在经过排序的连续容器上执行一个二进制搜索。这些算法与STL所提供的算法类似。

如果你已经非常熟悉STL容器并且在目标平台上STL是可用的,就没有必要再使用Qt容器了。有关STL类和函数的更多信息,对初学者来讲,SGI的STL网站http://www.sgi.com/tech/stl/是一个好地方。

在这一章中,我们还将关注QString、QByteArray和QVariant,因为它们与容器有很多相似之处。QString是贯穿Qt应用编程接口的一个16位Unicode字符串,QByteArray是一个用来存储原始二进制数据的8位字符数组,QVariant类则可以存储绝大多数C++和Qt值类型。

11.1 连续容器

QVector <T>是一种与数组相似的数据结构,它可以把项存储到内存中相邻近的位置,如图11.1所示。向量与普通C++数组的区别在于:向量知道自己的大小并且可以被重新定义大小。在向量末尾添加额外的项是非常快速有效的,而在向量前面或者中间插入项则是比较耗时的。

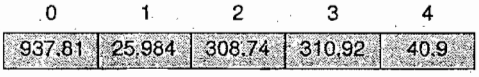


图11.1 双精度向量

如果能预先知道需要使用多少项,则在定义向量时,就可以初始化向量的大小,并使用[]操作符为它的项赋值;否则,可以稍后重新定义向量的大小,或者在向量的末端增加项。下面是一个指定向量初始大小的例子:

```
QVector<double> vect(3);
vect[0] = 1.0;
vect[1] = 0.540302;
vect[2] = -0.416147;
```

下面这个例子与上面的例子功能相同,只不过它是以一个空向量开始并使用 append()函数在这个向量的末端增加项:

```
QVector<double> vect;
vect.append(1.0);
vect.append(0.540302);
vect.append(-0.416147);
```

也可以使用 << 操作符来代替 append():

```
vect << 1.0 << 0.540302 << -0.416147;
```

而遍历向量的项的方式之一就是使用[]操作符和 count()函数:

```
double sum = 0.0;
for (int i = 0; i < vect.count(); ++i)
    sum += vect[i];
```

所创建的向量元素如果没有赋以确切的值,就会被使用这个项的类的默认构造函数进行初始化。基本类型和指针类型都会被初始化为 0。

对于较大的向量来说,在 QVector <T> 的开头或者中间插入项,或者在这些位置去除项,都是非常耗时的。因此,Qt 还提供了 QLinkedList <T> ,这是一种把项存储到内存中不相邻位置的数据结构,如图 11.2 所示。与向量不同,链表不支持快速的随机访问,但它提供了"常量时间"(constant time)的插入和删除。



图 11.2 双精度链表

链表并未提供[]操作符,所以必须使用迭代器来遍历项。迭代器还可用来指定项的位置。例如,下面的代码给出了如何在"Clash"和"Ramones"之间插入字符串"Tote Hosen":

```
QLinkedList<QString> list;
list.append("Clash");
list.append("Ramones");

QLinkedList<QString>::iterator i = list.find("Ramones");
list.insert(i, "Tote Hosen");
```

在本节的稍后部分,还将详细地研究迭代器。

QList <T> 连续容器是一个"数组列表"(array-list),结合了单一类中 QVector <T> 和 QLinkedList <T> 的最重要的优点。它支持随机访问,而且它的界面与 QVector 的一样是基于索引的。在 QList <T> 的任意一端插入或者移除项都是非常快速的,并且对含 1000 项以上的列表来说,在中间插入项也是很快的。除非我们想在一个极大的列表中执行插入或者要求列表中的元素都必须占据连续的内存地址,否则 QList <T> 通常是最合适采用的多用途容器类。

QStringList 类是被广泛用于 Qt 应用编程接口的 QList <QString> 的子类。除了从它的基类中继承的函数以外,QStringList 还提供一些特别的函数,以使得这种类对字符串的处理方式更通用。QStringList 将会在本章的最后一节中讨论到。

QStack <T> 和 QQueue <T> 是这些方便的子类中的另外两个例子。QStack <T> 是一个可以提供 push()、pop()和 top()的向量。QQueue <T> 是一个可以提供 enqueue()、dequeue()和 head()的列表。

对于目前所讲过的所有容器类,值类型 T 可以是一个与 int、double、指针类型、具有默认构造函数的类(没有参数的构造函数)、复制构造函数或者赋值操作符相似的类。符合这个条件的类包括:QByteArray、QDateTime、QRegExp、QString 和 QVariant。派生自 QObject 的 Qt 类不具备资格,因为

它们没有复制构造函数和赋值操作符。这在实际应用中并不是问题,因为可以简单地存储指向 QObject 类的指针而不是对象本身。

值类型 T 也可以是一个容器。在这种情况下,必须记得用空格分开连续的尖括号;否则,编译器将会把连续的尖括号认作是 >> 操作符而停止工作。例如:

```
QList<QVector<double> > list;
```

除了刚刚提到的类型,一个容器的值类型可以是符合之前所描述标准的任意自定义类。下面是这种类的一个例子:

```
class Movie
{
public:
    Movie(const QString &title = "", int duration = 0);

    void setTitle(const QString &title) { myTitle = title; }
    QString title() const { return myTitle; }
    void setDuration(int duration) { myDuration = duration; }
    QString duration() const { return myDuration; }

private:
    QString myTitle;
    int myDuration;
};
```

这个类有一个不需任何参数的构造函数(尽管它可以占用到两个参数)。它也有一个复制构造函数和一个赋值操作符,两个都是由 C++ 隐含提供的。对于这个类,逐项的复制就足够了,所以这里没有必要实现自己的复制构造函数和赋值操作符。

Qt 提供的两类迭代器用于遍历存储在容器中的项:Java 风格的迭代器和 STL 风格的迭代器。Java 风格的迭代器易于使用,STL 风格的迭代器则可以结合 Qt 和 STL 的一般算法而具有更加强大的功能。

对于每个容器类,都有两种 Java 风格的迭代器类型:只读迭代器和读-写迭代器。它们的有效位置如图 11.3 所示。只读迭代器类有 QVectorIterator <T> 、QLinkedListIterator <T> 和 QListIterator <T> 。相应的读-写迭代器则在其名字中都含有“Mutable”的字样(例如,QMutableVectorIterator <T>)。在讨论中,我们将重点关注 QList 的迭代器,针对链表和向量的迭代器都有相同的应用编程接口。

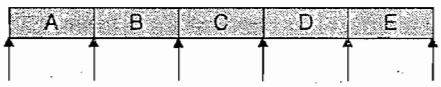


图 11.3 Java 风格迭代器的有效位置

当使用 Java 风格的迭代器时,必须首先牢记的是:它们本身并不是直接指向项的,而是能够定位在第一项之前、最后一项之后或者是两项之间。一个典型的迭代循环如下所示:

```
QList<double> list;
...
QListIterator<double> i(list);
while (i.hasNext()) {
    do_something(i.next());
}
```

迭代器通过容器遍历来初始化。在这一点,迭代器在第一项之前就被定位了。如果迭代器的右边有一个项,则调用 hasNext()函数会返回 true。next()函数返回迭代器右边的项并且将迭代器提升至下一有效位置。

向后迭代与此类似,但必须首先调用 toBack(),以将迭代器定位到最后一项之后的位置。

```
QListIterator<double> i(list);
i.toBack();
while (i.hasPrevious()) {
    do_something(i.previous());
}
```

如果迭代器的左边有一个项,则 hasPrevious()函数将返回 true。previous()函数返回迭代器左边的项并且将迭代器往前移一个位置。其实可以以另一种方式来认识 next()和 previous()迭代器,即它们返回迭代器跳过的项,如图 11.4 所示。

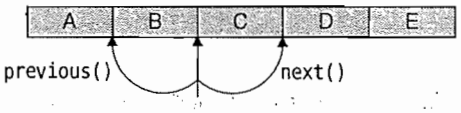


图 11.4 previous()和 next()对 Java 风格的影响

Mutable 迭代器(即 Java 风格的读-写迭代器)在遍历时提供了插入、修改以及删除项的函数。下面的循环删除了一个列表中所有的负数:

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    if (i.next() < 0.0)
        i.remove();
}
```

remove()函数总是对最后被跳过的项进行操作。在向后迭代时它也同样有效:

```
QMutableListIterator<double> i(list);
i.toBack();
while (i.hasPrevious()) {
    if (i.previous() < 0.0)
        i.remove();
}
```

与此类似,Java 风格的读-写迭代器提供 setValue()函数以修改最后被跳过的项。下面是使用绝对值来代替负值的代码:

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    int val = i.next();
    if (val < 0.0)
        i.setValue(-val);
}
```

通过调用 insert(),可以在当前迭代器位置插入一项。然后,迭代器就被提升到新的项和随后的项之间的位置。

除了 Java 风格的迭代器,每一个连续容器类 C <T> 都有两个 STL 风格的迭代器类型:C <T> :: iterator 和C <T> ::const_iterator。这两者的区别在于const_iterator不允许修改数据。

容器的 begin()函数返回引用容器中第一项的 STL 风格的迭代器(例如:list[0]),而 end()函数返回引用"最后一个项之后的"项的迭代器(例如,对于一个大小为 5 的列表取 list[5])。图 11.5 给出了 STL 风格的迭代器的有效位置。如果某个容器为空,则 begin()等价于 end()。这可以用来检查容器中是否有项,尽管以调用 isEmpty()函数的方式可更方便地达到这个目的。

STL 风格的迭代器的语法是模仿 C++ 数组的指针。我们可以使用 ++ 和 -- 操作符来移动下一项或者前一项,而使用一元操作符 * 来获得当前项。对于 QVector <T> ,iterator 和 const_iterator类型都只是 T * 和 const T * 的类型定义。(因为 QVector <T> 在连续的存储单元中存储它的项,所以这样做是可以的。)

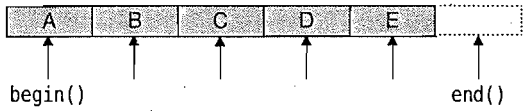


图 11.5　STL 风格迭代器的有效位置

下面的例子采用绝对值取代了 QList <double> 中的每个数值：

```
QList<double>::iterator i = list.begin();
while (i != list.end()) {
    *i = qAbs(*i);
    ++i;
}
```

一些 Qt 函数返回一个容器。如果想使用 STL 风格的迭代器遍历某个函数的返回值，则必须复制此容器并且遍历这个副本。例如，下面的代码给出了如何遍历由 QSplitter::sizes()返回的 QList <int> 的正确方式：

```
QList<int> list = splitter->sizes();
QList<int>::const_iterator i = list.begin();
while (i != list.end()) {
    do_something(*i);
    ++i;
}
```

下面的代码是错误的：

```
// WRONG
QList<int>::const_iterator i = splitter->sizes().begin();
while (i != splitter->sizes().end()) {
    do_something(*i);
    ++i;
}
```

这是因为每次调用 QSplitter::sizes()都返回一个新的 QList <int> 值。如果不存储这个返回值，则 C++ 在开始迭代之前就自动将其销毁，而只留下一个浮动迭代器。更糟糕的是，每次循环运行的时候，由于调用了 splitter -> sizes().end()，QSplitter::sizes()都必须生成一个新的列表的副本。总之，当使用 STL 风格的迭代器时，总是在返回值的容器副本上进行遍历。

利用只读 Java 风格的迭代器，不必复制容器。这个迭代器在后台自动生成一个副本，以确保总是遍历首先返回的函数的数据。例如：

```
QListIterator<int> i(splitter->sizes());
while (i.hasNext()) {
    do_something(i.next());
}
```

像这样复制一个容器看起来似乎耗费比较大，实际上不然，这是由于采用了名为隐含共享的最优化过程。这意味着复制一个 Qt 容器的速度大致就像复制一个简单指针一样快。只有在复制项之一发生改变时，数据才会实际被复制，而且这一切操作都可以在后台自动处理。由于这个原因，隐含共享(implicit sharing)有时候也称为“写时复制”。

隐含共享的优点在于它是一个我们不必考虑的最优化过程。它工作简单，不需要程序员的任何干预。同时，隐含共享提倡由值返回的对象的整洁的编程风格。考虑下面的函数：

```
QVector<double> sineTable()
{
    QVector<double> vect(360);
    for (int i = 0; i < 360; ++i)
        vect[i] = std::sin(i / (2 * M_PI));
    return vect;
}
```

该函数的调用看起来如下：

```
QVector<double> table = sineTable();
```

比较起来，STL 鼓励我们将向量作为一个非常量参数来传递，以避免当函数返回值被存储于变量中时发生复制：

```
void sineTable(std::vector<double> &vect)
{
    vect.resize(360);
    for (int i = 0; i < 360; ++i)
        vect[i] = std::sin(i / (2 * M_PI));
}
```

于是，这个调用程序就变得更加冗长，并且可读性较差：

```
std::vector<double> table;
sineTable(table);
```

Qt 对所有的容器和许多其他类都使用隐含共享，包括 QByteArray、QBrush、QFont、QImage、QPixmap 和 QString。这使得这些类不论是作为函数参数还是作为返回值，都可以非常有效地传递。

隐含共享是 Qt 对不希望修改的数据决不进行复制的保证。为了使隐含共享的作用发挥得最好，可以采取两个新的编程习惯。第一种习惯是对于一个(非常量的)向量或者列表进行只读存取时，使用 at()函数而不用[]操作符。因为 Qt 的容器类不能辨别[]操作符是否将出现在一个赋值的左边还是右边，它假设最坏的情况出现并且强制执行深层复制，而 at()函数则不被允许出现在一个赋值的左边。

当使用 STL 风格的迭代器遍历容器类的时候，类似的问题也将出现。只要在非常量的容器类上调用 begin()或 end()函数，并且如果数据是共享的，Qt 就会强制执行深层复制。为了防止这种低效操作的发生，一种解决的办法是无论何时都尽可能地使用 const_iterator、constBegin()和 constEnd()。

Qt 还提供了最后一种在连续容器中遍历项的方式——foreach 循环，如下所示：

```
QLinkedList<Movie> list;
...
foreach (Movie movie, list) {
    if (movie.title() == "Citizen Kane") {
        std::cout << "Found Citizen Kane" << std::endl;
        break;
    }
}
```

foreach 伪关键字按照标准的 for 循环实现。在循环的每一次迭代中，迭代变量(movie)都被设置为一个新项，从容器中的第一项开始向前迭代。foreach 循环会在进入循环时自动复制一个容器，因此即使在迭代过程中修改了容器类，也不会影响到循环。

break 和 continue 循环声明也是支持的。如果主体部分只由一个声明组成，则花括号是多余的。就如同 for 循环的声明语句一样，可以在循环体外定义迭代变量，具体如下：

```
QLinkedList<Movie> list;
Movie movie;
...
foreach (movie, list) {
    if (movie.title() == "Citizen Kane") {
        std::cout << "Found Citizen Kane" << std::endl;
        break;
    }
}
```

在循环体外定义迭代变量只对那些支持含有一个逗号的数据类型的容器类才适用(例如，QPair <QString,int>)。

隐含共享是如何工作的

隐含共享在后台自动运行，所以我们不必再编写任何代码来促使这个优化过程发生。但弄明白它到底是如何工作，的确是一件有益的事情。为此，我们将研究一个例子，看看在它的神秘面纱下到底发生了什么。这个例子使用了 QString，它是 Qt 众多的隐含共享类之一。

```
QString str1 = "Humpty";
QString str2 = str1;
```

我们设置 str1 为“Humpty”并令 str2 等于 str1。在这一点上，QString 的两个对象都指向内存中相同的内部数据结构。与字符数据一起，数据结构保存一个引用计数，以指出有多少 QString 指向相同的数据结构。因为 str1 和 str2 都指向相同的数据，所以引用计数的值为 2。

```
str2[0] = 'D';
```

当修改 str2 时，它首先将对数据进行深层复制，以确保 str1 和 str2 指向不同的数据结构，然后才将新数值应用于它所复制的数据。str1 的数据(“Humpty”)的引用计数变为 1，且把 str2 的数据(“Dumpty”)的引用计数也设为 1。引用计数为 1 表示数据并未被共享。

```
str2.truncate(4);
```

如果再次修改 str2，则由于 str2 数据的引用计数为 1，将不会发生数据复制。truncate()函数直接对 str2 的数据进行操作，导致字符串变为“Dump”。引用计数保持为 1。

```
str1 = str2;
```

当将 str2 赋给 str1 时，str1 的数据的引用计数降为 0，这意味着没有一个 QString 仍在使用“Humpty”数据。这样，这些数据就从内存中释放。两个 QString 都指向“Dump”，现在它的引用计数就是 2 了。

由于引用计数中的竞争情况，数据共享在多线程程序中通常只是作为一个选项而没有给与关注。使用 Qt，这并不是一个问题。在内部，容器类使用汇编语言指令执行基本的引用计数。通过 QSharedData 和 QSharedDataPointer 类，Qt 的用户也可以使用这项技术。

11.2 关联容器

关联容器可以保存任意多个具有相同类型的项，且它们由一个键索引。Qt 提供两个主要的关联容器类：QMap <K, T> 和 QHash <K, T> 。

QMap <K, T> 是一个以升序键顺序存储键值对的数据结构，如图 11.6 所示。这种排列使它可以提供良好的查找和插入性能以及键序的迭代。在内部，QMap <K, T> 是作为一个跳越列表(skip-list)来实现执行的。

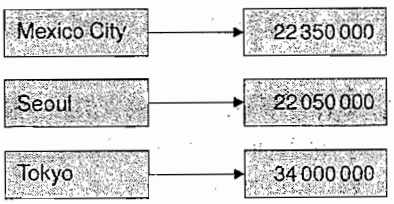


图 11.6 QString 到 int 的映射

在映射中插入项的一种简单方式是调用 insert():

```
QMap<QString, int> map;
map.insert("eins", 1);
map.insert("sieben", 7);
map.insert("dreiundzwanzig", 23);
```

另外,也可以像下面一样,给一个指定的键赋值:

```
map["eins"] = 1;
map["sieben"] = 7;
map["dreiundzwanzig"] = 23;
```

[]操作符既可以用于插入也可以用于检索。如果在非常量映射中使用[]为一个不存在的键检索值,则会用给定的键和空值创建一个新的项。为了避免意外地创建空值,可以使用 value()函数代替[]操作符来获得项。

```
int val = map.value("dreiundzwanzig");
```

如果键不存在,则利用值类型的默认构造函数,将返回一个默认值,同时不会创建新的项。对于基本类型和指针类型,将返回 0 值。我们可以指定另一默认值作为 value()的第二个参数,例如:

```
int seconds = map.value("delay", 30);
```

这相当于:

```
int seconds = 30;
if (map.contains("delay"))
    seconds = map.value("delay");
```

QMap <K,T> 的 K 和 T 数据类型可以是与 int、double、指针类型、有默认构造函数的类、复制构造函数和赋值操作符相似的基本数据类型。此外,K 类型必须提供 operator < (),因为 QMap <K,T> 要使用这个操作符以升键序顺序存储项。

QMap <K,T> 有一对方便的函数 keys()和 values(),它们在处理小数据集时显得特别有用。它们分别返回映射键的 QList 和映射值的 QList。

映射通常都是单一值的:如果赋予一个现有的键一个新值,则原有的旧值将被该新值取代,以确保两个项不会共有同一个键。通过使用 insertMulti()函数或者 QMultiMap <K,T> 方便的子类,可以让多个键值对有相同的键。QMap <K,T> 重载了 values(const K &),返回一个给定键所有值的 QList 列表。例如:

```
QMultiMap<int, QString> multiMap;
multiMap.insert(1, "one");
multiMap.insert(1, "eins");
multiMap.insert(1, "uno");

QList<QString> vals = multiMap.values(1);
```

QHash <K,T> 是一个在哈希表中存储键值对的数据结构。它的接口几乎与 QMap <K,T> 相同,但是与 QMap <K,T> 相比,它对 K 的模板类型有不同的要求,而且它提供了比 QMap <K,T> 更快的查找功能。

除了对存储在容器类中的所有值类型的一般要求,QHash <K,T> 中 K 的值类型还需要提供一个 operator == (),并需要一个能够为键返回哈希值的全局 qHash()函数的支持。Qt 已经为 qHash()函数提供了整型、指针型、QChar、QString 以及 QByteArray。

QHash <K,T> 为它内部的哈希表自动分配最初的存储区域,并在有项被插入或者删除时重新划分所分配的存储区域的大小。也可以通过调用 reserve()或者 squeeze()来指定或者压缩希望存

储到哈希表中的项的数目,以进行性能调整。通常的做法是利用我们预期的最大的项的数目来调用 reserve(),然后插入数据,最后如果有多出的项,则调用 squeeze()以使内存的使用减到最小。

虽然哈希表通常都是单一值的,但是使用 insertMulti()函数或者 MultiHash <K,T> 方便的子类,也可以将多个值赋给同一个键。

除了 QHash <K,T> 之外,Qt 还提供了一个用来高速缓存与键相关联的对象的 QCache <K,T> 类以及仅仅存储键的 QSet <K> 容器。在内部,它们都依赖于 QHash <K,T> ,且都像 QHash <K,T> 一样对 K 的类型有相同的要求。

最简便的遍历存储在关联容器中所有键值对的方式是使用 Java 风格的迭代器。因为迭代器必须能同时访问键和值,针对关联容器的 Java 风格的迭代器与连续容器的在运作方式有些差异。主要的区别在于 next()和 previous()函数返回一个代表键值对的对象,而不是一个简单的值。我们可以使用 key()和 value()分别从这个对象中获得键和值。例如:

```
QMap<QString, int> map;
...
int sum = 0;
QMapIterator<QString, int> i(map);
while (i.hasNext())
    sum += i.next().value();
```

如果需要同时存取键和值,可以先忽略 next()或 previous()的返回值并使用迭代器的 key()和 value()函数,它们都是针对最后被跳过的项进行操作的:

```
QMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() > largestValue) {
        largestKey = i.key();
        largestValue = i.value();
    }
}
```

Mutable 迭代器(即 Java 风格的读-写迭代器)拥有可以修改与当前项相关联的值的 setValue()函数:

```
QMutableMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() < 0.0)
        i.setValue(-i.value());
}
```

STL 风格的迭代器也提供了 key()和 value()函数。对于非常量迭代器类型,value()返回一个允许在迭代时改动其数值的非常量参数。需要注意的是:尽管这些迭代器都被称为 STL 风格,它们却与基于 std::pair <K,T> 的 std::map <K,T> 迭代器有很大差别。

foreach 循环也可以用在关联容器中,但是它仅对键值对上的值分量有效。如果同时需要项中的键和值,可以在如下的嵌套式 foreach 循环中调用 keys()和 values(const K &)函数:

```
QMultiMap<QString, int> map;
...
foreach (QString key, map.keys()) {
    foreach (int value, map.values(key)) {
        do_something(key, value);
    }
}
```

11.3 通用算法

<QtAlgorithms> 的头文件声明了在容器类上实现基本算法的一套全局模板函数。这些函数中的大部分都是在 STL 风格上的迭代器上工作的。

STL 的 <algorithm> 头文件提供了一套更为完整的通用算法。这些算法既可以在 STL 容器类上使用,也可以在 Qt 容器类上使用。如果 STL 的实现代码在所有平台上都可以得到,那么在没有对应的 Qt 算法时,就没有理由不使用 STL 算法。这里将引入最重要的 Qt 算法。

qFind()算法在容器类中查找一个特定的值。它接受一个"begin"和一个"end"迭代器,并且返回一个与其匹配的指向第一项的迭代器;如果没有匹配的项,则返回"end"。在下面的例子中,将 i 设置为 list.begin() + 1,而 j 设置为 list.end()。

```
QStringList list;
list << "Emma" << "Karl" << "James" << "Mariette";

QStringList::iterator i = qFind(list.begin(), list.end(), "Karl");
QStringList::iterator j = qFind(list.begin(), list.end(), "Petra");
```

qBinaryFind()算法执行的搜索操作与 qFind()算法相似,其区别在于 qBinaryFind()算法假设项都是以升序的顺序存储的,并且使用了快速二分搜索而不是 qFind()算法的线性搜索。

qFill()算法采用一个特定的值组装一个容器:

```
QLinkedList<int> list(10);
qFill(list.begin(), list.end(), 1009);
```

与其他基于迭代器的算法类似,也可以通过变化参数以在一部分容器类上使用 qFill()。下面摘录的代码将一个向量的前 5 项初始化为 1009,将后 5 项初始化为 2013:

```
QVector<int> vect(10);
qFill(vect.begin(), vect.begin() + 5, 1009);
qFill(vect.end() - 5, vect.end(), 2013);
```

qCopy()算法将一个容器类的值复制到另一个容器类中:

```
QVector<int> vect(list.count());
qCopy(list.begin(), list.end(), vect.begin());
```

qCopy()也可以用来在同一个容器类中复制值,只要数据来源范围与目标范围不重叠。下面是一个代码摘录片段,它利用列表的前两个项复写该列表的后两项:

```
qCopy(list.begin(), list.begin() + 2, list.end() - 2);
```

qSort()算法则以升序排列容器类中的项:

```
qSort(list.begin(), list.end());
```

默认情况下,qSort()使用 < 操作符对项进行比较。为了以升序排列项,我们将 qGreater <T> () 作为第三个参数来传递(其中 T 为容器类的值类型),过程如下:

```
qSort(list.begin(), list.end(), qGreater<int>());
```

可以用第三个参数来定义用户的排序标准。例如,下面是一个"小于"比较函数,它以不区分大小写的方式比较了不同的 QString:

```
bool insensitiveLessThan(const QString &str1, const QString &str2)
{
    return str1.toLower() < str2.toLower();
}
```

于是,qSort()的调用变为:

```
QStringList list;
...
qSort(list.begin(), list.end(), insensitiveLessThan);
```

qStableSort()算法与qSort()很相似,但qStableSort()算法还可以保证进行对等比较的项在排序之后表现出与之前相同的顺序。如果排序标准仅仅考虑值的一部分并且用户可以看到结果,这就非常有用了。第4章曾使用了qStableSort()在Spreadsheet应用程序中实现了排序。

qDeleteAll()算法对每一个存储在容器类中的指针调用delete。这仅对于那些为指针类型的容器类才有意义的。在调用之后,这些项仍然作为悬摆指针存在于容器类上,因此我们通常也应该在容器类上调用clear()。例如:

```
qDeleteAll(list);
list.clear();
```

qSwap()算法可以交换两个变量的值,例如:

```
int x1 = line.x1();
int x2 = line.x2();
if (x1 > x2)
    qSwap(x1, x2);
```

最后,被所有其他的Qt首部都包括的<QtGlobal>的头文件,为我们提供了一些有用的定义,其中包括返回参数绝对值的qAbs()函数,以及返回最大、最小值的qMin()和qMax()函数。

11.4 字符串、字节数组和变量

QString、QByteArray和QVariant这三种类与容器类在许多方面都有相同之处,在一些情况下都可以用来代替容器类的使用。与容器类相似,这些类也是使用隐含共享来最优化内存和速度。

我们将从QString开始。每一个图形用户界面(GUI)程序都会用到字符串,不仅仅是为用户界面,更多的是为数据结构所用。C++本身提供两种字符串:传统的C语言型的以"\0"结尾的字符数组和std::string类。与这两种字符串不同,QString支持16位Unicode值。Unicode码以ASCII码和Latin-1码为子集,具有它们常用的数字值。但由于QString是16位的,它可以表示数千种其他字符以表达世界上绝大多数的语言。关于Unicode的更多信息,可以参考第18章。

当使用QString时,我们不必担心那些诸如如何分配足够的内存空间或确保数据以"\0"结尾等晦涩难懂的细节。从概念上说,可以将QString看成QChar向量。QString可以嵌入"\0"字符。length()函数会返回包括被嵌入的"\0"字符的整个字符串的大小。

QString为连接两个字符串提供了一个二进制+操作符,还为在字符串后追加字符串提供了+=操作符。因为QString在行数据的末端自动预分配内存空间,因此通过重复添加字符来建立字符串就相当快捷了。下面是一个结合使用+和+=的例子:

```
QString str = "User: ";
str += userName + "\n";
```

以下的QString::append()函数与+=操作符的功能相同:

```
str = "User: ";
str.append(userName);
str.append("\n");
```

使用QString的sprintf()函数则是连接字符串的另一种完全不同的方法:

```
str.sprintf("%s %.1f%%", "perfect competition", 100.0);
```

这个函数支持与C++库sprintf()函数相同的格式说明符。在之前的例子中,str被赋值为"perfect competition 100.0%"。

从其他字符串或者数组来建立一个字符串的另一种方法是使用 arg():

```
str = QString("%1 %2 (%3s-%4s)")
      .arg("permissive").arg("society").arg(1950).arg(1970);
```

在这个例子中,"permissive"会取代"%1","society"会取代"%2","1950"会取代"%3",而"1970"会取代"%4"。相应的结果为"permissive society (1950s-1970s)"。arg()的重载可以处理各种数据类型。对于控制字段长度、数值基数或者浮点精度等,一些重载有额外的参数。相较于 sprintf()而言,arg()通常是一个更好的解决方案,因为是它类型安全的,完全支持 Unicode 编码,并且允许译码器对"%n"参数进行重新排序。

通过使用 QString::number()静态函数,QString 可以将数字转换为字符串:

```
str = QString::number(59.6);
```

或者也可以使用 setNum()函数:

```
str.setNum(59.6);
```

还可以使用 toInt()、toLongLong()、toDouble()等来完成从字符串到数字的逆转换。例如:

```
bool ok;
double d = str.toDouble(&ok);
```

这些函数接受一个任选的指向 bool 变量的指针,并且根据转换成功与否来将变量值设为 true 或 false。如果转换没有完成,这些函数将返回 0。

一旦有了字符串,我们通常都希望从字符串中提取出所需的部分。mid()函数返回在给定位置(第一个参数)开始且达到给定长度(第二个参数)的子串。例如,下面的代码在控制台上打印出"pays"[①]:

```
QString str = "polluter pays principle";
qDebug() << str.mid(9, 4);
```

如果省略第二个参数,mid()函数返回在给定位置开始到字符串末端结束的子串。例如,下面的代码在控制台上打印出"pays principle":

```
QString str = "polluter pays principle";
qDebug() << str.mid(9);
```

也可以使用 left()和 right()函数来完成相似的任务。它们俩都可以接收字符的数量 n,并且返回字符串的前 n 个字符或最后 n 个字符。例如,下面的代码在控制台上打印出"polluter principle":

```
QString str = "polluter pays principle";
qDebug() << str.left(8) << " " << str.right(9);
```

如果想查明一个字符串是否包含一个特定的字符、子串或者正则表达式,可以使用 QString 中的 indexOf()函数:

```
QString str = "the middle bit";
int i = str.indexOf("middle");
```

这会将 i 设置为 4。在失败时,indexOf()函数返回 -1,并且接收一个可以的开始位置和区分大小写的标记。

如果仅仅想要检查字符串是否以某个字符(串)开始或者结束,则可以使用 startsWith()和 endsWith()函数:

① 这里用到的方便的 qDebug() << arg 语法要求包含头文件(<QtDebug>),而 qDebug("...",arg)语法在任意一个包含至少一个 Qt 首部的文件中都是可用的。

```
if (url.startsWith("http:") && url.endsWith(".png"))
    ...
```

上面的代码就比下面的代码更简单、快捷：

```
if (url.left(5) == "http:" && url.right(4) == ".png")
    ...
```

利用 == 操作符进行字符串比较是区分大小写的。如果要比较那种用户可见的字符串，localeAwareCompare()函数通常是一个不错的选择；如果比较并不区分大小写，则可以使用 toUpper()或 toLower()函数，例如：

```
if (fileName.toLower() == "readme.txt")
    ...
```

如果想使用一个字符串来代替另一个字符串中的某一部分，可以使用 replace()：

```
QString str = "a cloudy day";
str.replace(2, 6, "sunny");
```

字符串的结果是“a sunny day”。上面的代码可以用 remove()和 insert()重新写为：

```
str.remove(2, 6);
str.insert(2, "sunny");
```

首先，我们去掉从第二个位值开始的 6 个字符，字符串因此变为“a day”（含有两个空格），然后在第二个位置插入“sunny”。

replace()函数的重载版本让第二个参数代替所有第一个参数出现的地方。例如，下面是使用“&”代替字符串中所有的“&”：

```
str.replace("&", "&amp;");
```

我们经常需要删除一个字符串中空白处的空格（比如空格符、制表符、换行符等）。QString 有一个可以从字符串的两端删除空白处的空格的函数：

```
QString str = "   BOB \t THE  \nDOG \n";
qDebug() << str.trimmed();
```

字符串 str 描述如下：

			B	O	B		\t		T	H	E			\n	D	O	G		\n

由 trimmed()返回的字符串为：

B	O	B		\t		T	H	E			\n	D	O	G

当处理用户输入数据时，除了从字符串的两端删除空白处的空格符外，通常还想用简单的空格符代替字符串内部每一连续空白处的空格：

```
QString str = "   BOB \t THE  \nDOG \n";
qDebug() << str.simplified();
```

由 simplified()返回的字符串为：

B	O	B		T	H	E		D	O	G

使用 QString::split()，可以把一个字符串分成一些 QStringList 子串：

```
QString str = "polluter pays principle";
QStringList words = str.split(" ");
```

在上面的例子中，我们把字符串“polluter pays principle”分成了三个子串，“polluter”、“pays”和“principle”。split()函数有一个可选的第二参数，用来指定是否空的子串应该被保留（默认为保留）还是被删除。

使用 join()函数，QStringList 中的项可以连接起来形成一个单一的字符串。在每一对被连接的

字符串之间都要插入 join()的参数。例如,下面是如何创建一个由 QStringList 包含的所有字符串组成的、按字母表顺序排列且由换行符分隔的简单字符串:

```
words.sort();
str = words.join("\n");
```

当对字符串进行操作时,通常需要判断字符串是否为空。这可以通过调用 isEmpty()或者检查 length()是否为 0 来完成。

大多数情况下,从 const char * 字符串到 QString 的转换是自动完成的,例如:

```
str += " (1870)";
```

这里,我们采用无格式形式为 QString 加上一个 const char * 。为了明确地将 QString 转换为 const char * ,可以只使用一个 QString 强制转换,或者调用 fromAscii()或 fromLatin1()。(关于其他编码中的文字字符串处理的说明,可以参见第 18 章。)

要将 QString 转换为 const char * ,可以使用 toAscii()或 toLatin1()函数,它们返回 QByteArray,而利用 QByteArray::data()或 QByteArray::constData(),可以将 QByteArray 转换为 const char * 。例如:

```
printf("User: %s\n", str.toAscii().data());
```

为了方便起见,Qt 提供了 qPrintable()宏用来执行与序列 toAscii().constData()相同的功能:

```
printf("User: %s\n", qPrintable(str));
```

当在 QByteArray 上调用 data()或 constData()时,返回的字符串属于 QByteArray 对象。这就意味着不必为内存泄漏而担心了,Qt 将为我们重新收回内存。另一方面,必须注意不要太长时间地使用指针。如果 QByteArray 没有存储在一个变量中,那么它将在声明的末端自动删除。

QByteArray 类有一个与 QString 很相似的应用编程接口。诸如 left()、right()、mid()、toLower()、toUpper()、trimmed()和 simplified()等函数,在 QByteArray 中的语义形式与在 QString 中的相同。QByteArray 对于存储原始的二进制数据以及 8 位编码的文本字符串非常有用。一般说来,我们推荐使用 QString 而不是 QByteArray 来存储文本,因为 QString 支持 Unicode 编码。

为方便起见,QByteArray 自动保证"最后一个项之后的项"总为"\0",这使得利用 const char * 可以很容易地将 QByteArray 传递给一个函数。QByteArray 还支持嵌入的"\0"字符,以允许我们存储任意的二进制数据。

在某些情况下,我们需要在同一个变量中存储不同类型的数据。一种方法是像 QByteArray 或 QString 一样,对数据进行编码。例如,字符串可以支持文本值或者以字符串形式支持数字值。这些方法很灵活,但是它抛弃了 C++ 的一些优点,尤其是类型的安全性和效率。Qt 提供了一个更加灵巧的方法,也就是 QVariant,来处理那些能够支持不同数据类型的变量。

QVariant 类可以支持许多种 Qt 类型的值,除了基本的 C++ 数字类型(如 double 和 int),还包括 QBrush、QColor、QCursor、QDateTime、QFont、QKeySequence、QPalette、QPen、QPixmap、QPoint、QRect、QRegion、QSize 和 QString。QVariant 类也支持容器类,如 QMap <QString, QVariant> 、QStringList 和 QList <QVariant> 。

在项视图类、数据库模块和 QSettings 中广泛使用了变量,变量允许我们读写项数据、数据库数据以及任意与 QVariant 兼容的用户首选参数。第 3 章已经看到了这样的一个例子,在那个例子中,我们将 QRect、QStringList 和一对 bool 值以变量的形式传给了 QSettings::setValue(),并在之后以变量的形式将它们取回。

通过容器类的嵌套值,可以利用 QVariant 创建任意复杂的数据结构:

```
QMap<QString, QVariant> pearMap;
pearMap["Standard"] = 1.95;
pearMap["Organic"] = 2.25;

QMap<QString, QVariant> fruitMap;
fruitMap["Orange"] = 2.10;
fruitMap["Pineapple"] = 3.85;
fruitMap["Pear"] = pearMap;
```

这里,我们利用字符串键(产品名)和浮点数(价格)或者映射的值创建一个映射。顶级映射包括三个键:“Orange”、“Pear”和“Pineapple”。与“Pear”键相关联的值是一个包含两个键(“Standard”和“Organic”)的映射。当遍历一个支持变量值的映射时,需要使用 type()来检查变量保存所支持的类型,以便做出适当的反应。

创建这样的数据结构是非常吸引人的,因为能够以任意方式组织数据。但是 QVariant 的便利性是以降低效率及可读性为代价的。通常,定义一个适当的 C++ 类来存储随时可能的数据,是值得的。

Qt 的元对象系统使用 QVariant,因此它也是 QtCore 模块的一部分。但是,当与 QtGui 模块相连时,QVariant 可以存储与图形用户界面相关的类型,例如 QColor、QFont、QIcon、QImage 和 QPixmap:

```
QIcon icon("open.png");
QVariant variant = icon;
```

为了从 QVariant 中获得与图形用户界面相关的值,可以使用如下的 QVariant::value <T> ()模板成员函数:

```
QIcon icon = variant.value<QIcon>();
```

value <T> ()函数也可以用在非图形用户界面数据类型和 QVariant 之间进行转换,但实际上,对于非图形用户界面类型,通常使用 to...()作为非图形用户界面数据类型的转换函数[例如 toString()]。

如果自定义数据类型提供了默认的构造函数和副本构造函数的话,QVariant 也可以用来存储它们。为此,必须首先使用 Q_DECLARE_METATYPE()宏注册数据类型,尤其是在类定义下的头文件中:

```
Q_DECLARE_METATYPE(BusinessCard)
```

这使我们可以使用如下的方式来编写代码:

```
BusinessCard businessCard;
QVariant variant = QVariant::fromValue(businessCard);
...
if (variant.canConvert<BusinessCard>()) {
    BusinessCard card = variant.value<BusinessCard>();
    ...
}
```

由于编译器的局限性,在 MSVC 6 中,这些模板成员函数并不可用。如果需要用到这个编译器,可以用 qVariantFromValue()、qVariantValue <T> ()和 qVariantCanConvert <T> ()全局函数来代替。

如果自定义数据类型采用 << 和 >> 操作符来完成从 QDataStream 的读写,就可以使用 qRegisterMetaTypeStreamOperators <T> ()来注册这些自定义数据类型。这就可以利用 QSettings 在其他类型之中存储自定义数据类型的首选参数。例如:

```
qRegisterMetaTypeStreamOperators<BusinessCard>("BusinessCard");
```

除了讨论 QString、QByteArray 和 QVariant 外,本章还重点论述了 Qt 容器类。除了这些类以外,Qt 还提供了一些其他的容器类。QPair <T1,T2> 是其中之一,它与 std::pair <T1,T2> 很相似,可以

简单地存储两个值。另外一个就是 QBitArray,21.1 节将用到它。最后,还有 QVarLengthArray <T, Prealloc> ,它是 QVector <T> 的另一低级候选方案。因为 QVarLengthArray <T, Prealloc> 在堆栈中预分配内存空间且它不是隐含共享的,所以它的系统开销比 QVector <T> 更少,这使它更适合于紧凑的小循环。

Qt 的所有算法,包括一些这里没有介绍到的 qCopyBackward()和 qEqual()等等,都在 Qt 的在线文档资料中有相关的描述。要想知道更多的关于 Qt 容器类的信息,可以参考 http://doc.trolltech.com/4.3/algorithms.html;而那些关于它们的空间复杂度和增长策略的信息,可以参考 http://doc.trolltech.com/4.3/containers.html。

第 12 章　输入与输出

从文件或者其他设备读取或者写入数据几乎是每个应用程序共有的特点。Qt 通过 QIODevice 为输入输出提供了极佳的支持。QIODevice 是一个封装能够读写字节块“设备”的强有力的提取器。Qt 包括如下的 QIODevice 子类：

QFile	在本地文件系统和嵌入式资源中存取文件
QTemporaryFile	在本地文件系统中创建并存取临时文件
QBuffer	从 QByteArray 中读取或者写入数据
QProcess	运行外部程序并处理进程间通信
QTcpSocket	利用 TCP 在网络上传输数据流
QUdpSocket	在网络上发送或接收 UDP 数据报
QSslSocket	利用 SSL/TLS 在网络上传输加密数据流

QProcess、QTcpSocket、QUdpSocket 和 SslSocket 都是顺序存储设备，这意味着所存储的数据从第一个字节开始到最后一个字节为止只能被读取一次。QFile、QTemporaryFile 和 QBuffer 则是随机存取设备，因此可以从任意位置多次读取字节位所存储的数据。这些随机存取设备还提供了 QIODevice::seek()函数以重新配置文件指针。

除了设备类，Qt 还提供了两个更高级别的流类，使我们可以从任意的输入输出设备读取或写入数据：QDataStream 用来读写二进制数据，QTextStream 用来读写文本数据。这些类考虑了诸如字节顺序与文本编码等方面的问题，以保证运行在不同平台或者不同语言环境下的 Qt 应用程序可以相互读写文件。这就使得 Qt 的输入输出类相较于对应的标准 C++ 类来说，变得更加方便了，后者通常是将字节排序与文本编码等问题留给了程序员来处理的。

QFile 使存取单个文件变得简单，不论它们是在文件系统中还是作为资源嵌入在应用程序的可执行文件中。对于需要依靠整个文件组集一同运作的应用程序，Qt 提供了 QDir 和 QFileInfo 类，它们分别用于处理目录地址和提供内部文件信息。

QProcess 类允许启动外部程序并通过标准输入、输出以及标准错误通道(cin、cout 和 cerr)与外部程序交互。可以设置外部程序的环境变量和工作目录。在默认情况下，交互的执行过程是异步的(不阻塞的)，但是不排除在某些操作上发生阻塞的可能。

网络与 XML 的读写，都是非常重要的主题，关于它们的内容将在专门的章节(第 15 章和第 16 章)中分别介绍。

12.1　读取和写入二进制数据

Qt 中载入和保存二进制数据的最简单方式是通过实例化一个 QFile 打开文件，然后通过 QDataStream 对象存取它。QDataStream 提供了一种与运行平台无关的存储格式，它不仅支持 QList <T> 和 QMap <K,T> 等 Qt 容器类，还支持整型和双精度型等基本的 C++ 类，以及其他许多种 Qt 数据类型，诸如 QByteArray、QFont、QImage、QPixmap、QString 和 QVariant。

下面是如何在一个名为 facts.dat 的文件中存储一个整型数据、一个 QImage 以及一个 QMap <QString,QColor> 的代码:

```
QImage image("philip.png");

QMap<QString, QColor> map;
map.insert("red", Qt::red);
map.insert("green", Qt::green);
map.insert("blue", Qt::blue);

QFile file("facts.dat");
if (!file.open(QIODevice::WriteOnly)) {
    std::cerr << "Cannot open file for writing: "
              << qPrintable(file.errorString()) << std::endl;
    return;
}

QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_3);

out << quint32(0x12345678) << image << map;
```

如果不能打开文件,就通知用户并返回。qPrintable()宏将为 QString 返回一个 const char * 。[另外一种方法是使用 QString::toStdString(),它返回 std::string, <iostream> 将为其重载一个 << 。]

如果成功打开文件,就创建 QDataStream,同时设置它的版本号。版本号是一个整数,它会影响 Qt 数据类型的表示方式(基本的 C++ 数据类型的表示方式总是相同的)。在 Qt 4.3 中,最全面的版本号是第 9 版(version 9)。我们可以手动输入常数 9 或者使用 QDataStream::Qt_4_3 符号名。

为了确保数字 0x12345678 在所有平台上都是按照无符号 32 位整数形式写入的,我们将它强制转换为 quint32,这是一个严格保证为 32 位的数据类型。为了保证互通性,QDataStream 会默认将高字节在后的顺序(big-endian)作为标准,这可以通过调用 setByteOrder()来改变。

我们不必明确地关闭这个文件,因为当 QFile 变量离开它的作用域时,文件会自动关闭。如果想检验数据是否被真正写入,可以调用 flush()并检查其返回值(若返回值为 true,则表示成功写入数据)。

读回数据的代码借鉴了用以写入数据的代码:

```
quint32 n;
QImage image;
QMap<QString, QColor> map;

QFile file("facts.dat");
if (!file.open(QIODevice::ReadOnly)) {
    std::cerr << "Cannot open file for reading: "
              << qPrintable(file.errorString()) << std::endl;
    return;
}

QDataStream in(&file);
in.setVersion(QDataStream::Qt_4_3);

in >> n >> image >> map;
```

用于读取数据的 QDataStream 版本与用于写入数据的 QDataStream 版本一样。必须保证这一点。通过手动编写版本号,可以确保应用程序可以正常读写数据(假设它是由 Qt 4.3 或者更新的 Qt 版本进行编译的)。

QDataStream 这种存储数据的方式使我们可以完全连续地读回数据。例如,QByteArray 表示一个由它们自己字节数尾随的 32 位的字节计数。利用 readRawBytes()和 writeRawBytes(),QDataStream 也可以用来读写一些原始的二进制数据而不需要任何的字节计数首部。

当从 QDataStream 读取数据时,错误处理相当容易。流有一个 status()值,可以是 QDataStream::Ok、

QDataStream::ReadPastEnd 或者 QDataStream::ReadCorruptData。如果错误发生,则 >> 操作符总是读取 0 值或者空值。这表示通常可以简单地读取一个文件而不用担心出错,并在最后检查status()值以确定读取的文件数据有效。

QDataStream 可以处理多种 C++ 和 Qt 数据类型,完整的列表可以从网站 http://doc.trolltech.com/4.3/datastreamformat.html 获得。还可以通过重载 << 和 >> 操作符为用户的自定义类型增加支持。以下是可用于 QDataStream 的自定义数据类型的定义:

```
class Painting
{
public:
    Painting() { myYear = 0; }
    Painting(const QString &title, const QString &artist, int year) {
        myTitle = title;
        myArtist = artist;
        myYear = year;
    }

    void setTitle(const QString &title) { myTitle = title; }
    QString title() const { return myTitle; }
    ...

private:
    QString myTitle;
    QString myArtist;
    int myYear;
};

QDataStream &operator<<(QDataStream &out, const Painting &painting);
QDataStream &operator>>(QDataStream &in, Painting &painting);
```

下面的代码说明了如何实现 << 操作符:

```
QDataStream &operator<<(QDataStream &out, const Painting &painting)
{
    out << painting.title() << painting.artist()
        << quint32(painting.year());
    return out;
}
```

为了输出 Painting,我们简单地输出两个 QString 和一个 quint32。在函数的最后,返回这个流。这是一个常见的允许以一个输出流来使用一系列 << 操作符的 C++ 习惯用法。

```
out << painting1 << painting2 << painting3;
```

operator >> ()的实现与 operator << ()的实现相似:

```
QDataStream &operator>>(QDataStream &in, Painting &painting)
{
    QString title;
    QString artist;
    quint32 year;

    in >> title >> artist >> year;
    painting = Painting(title, artist, year);
    return in;
}
```

为自定义数据类型提供流操作符有几个好处。其中之一是允许流输出使用自定义类型的容器类。例如:

```
QList<Painting> paintings = ...;
out << paintings;
```

还可以在容器类中轻松读取:

```
QList<Painting> paintings;
in >> paintings;
```

如果 Painting 不支持 << 或 >> 操作符,则将产生编译器错误。为自定义数据类型提供流操作符的另一个好处是:可以将这些数据类型的值存储为 QVariant 的形式,这便于它们在更大的范围内使用,例如通过 QSettings。这些工作都是假设预先使用 qRegisterMetaTypeStreamOperators <T> ()注册了数据类型,如第 11 章(见 221 页)中所说明的。

当用到 QDataStream 时,Qt 考虑了读取和写入的每一个数据类型,包括具有任意多的项的容器类。这使我们不必再组织排列所写入的数据,也不必再对所读入的数据进行解析。我们唯一的任务是确保读取各种数据类型时的顺序与写入时的严格一致,而细节上的问题就留给 Qt 去处理了。

QDataStream 对用户自定义应用程序文件格式和标准二进制数据格式都是有用处的。我们可以使用流式操作符或者 readRawBytes()和 writeRawBytes()在基本数据类型(如 quint16 或 float)上读取和写入标准二进制数据格式。如果 QDataStream 仅仅只用于读写基本 C++ 数据类型,甚至都不必调用 setVersion()函数。

到目前为止,我们使用硬编码的流版本号 QDataStream::Qt_4_3来载入和保存数据。这种方式是简单可靠的,但是它也有一个小缺点:不能利用新的版本或者更新的格式。例如,如果 Qt 的一个更新版本为 QFont 添加了一个新的属性(即除了它的点大小、族属性等之外的属性),而硬编码的版本号为 Qt_4_3,那么将不能保存或者载入这个属性。针对这个情况,有两种解决方案。第一种解决方案是把 QDataStream 版本号嵌入到文件中:

```
QDataStream out(&file);
out << quint32(MagicNumber) << quint16(out.version());
```

(其中,MagicNumber 表示唯一一个标识文件类型的常数)。这种方法可以确保在任何情况下都使用最新的 QDataStream 版本来写入数据。当读取文件时,将读取流版本:

```
quint32 magic;
quint16 streamVersion;

QDataStream in(&file);
in >> magic >> streamVersion;

if (magic != MagicNumber) {
    std::cerr << "File is not recognized by this application"
              << std::endl;
} else if (streamVersion > in.version()) {
    std::cerr << "File is from a more recent version of the "
              << "application" << std::endl;
    return false;
}

in.setVersion(streamVersion);
```

只要流版本小于或等于应用程序所使用的版本,就可以读取数据;否则报告一个错误。

如果文件格式包含一个它自己的版本号,则可以利用它来推断出流版本号而不是明确地存储它。例如,假设文件格式是针对应用程序的 1.3 版本,然后可以如下编写代码:

```
QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_3);
out << quint32(MagicNumber) << quint16(0x0103);
```

当回读数据时,可根据应用程序的版本号来决定所使用的 QDataStream 版本:

```
QDataStream in(&file);
in >> magic >> appVersion;

if (magic != MagicNumber) {
    std::cerr << "File is not recognized by this application"
```

```
                << std::endl;
        return false;
    } else if (appVersion > 0x0103) {
        std::cerr << "File is from a more recent version of the "
                << "application" << std::endl;
        return false;
    }

    if (appVersion < 0x0103) {
        in.setVersion(QDataStream::Qt_3_0);
    } else {
        in.setVersion(QDataStream::Qt_4_3);
    }
```

在这个实例中，我们规定由这个应用程序1.3版本之前的版本所保存的任何文件都使用数据流版本4(Qt_3_0)，而由这个应用程序的1.3版本保存的文件使用数据流版本9(Qt_4_3)。

总之，一共有三种处理QDataStream版本的策略：硬编码的版本号、明确地写入并读取版本号以及根据应用程序版本号使用不同的硬编码版本号。这些策略都可以用来确保由应用程序的较老版本写入的数据可以被新版本成功读取，即使这个新版本的应用程序与更新的Qt版本关联。一旦选择了一种处理QDataStream版本的策略，使用Qt读取和写入二进制数据时就变得简单且可靠。

如果想一次读取或者写入一个文件，可以完全不用QDataStream而使用QIODevice的write()和readAll()函数。例如：

```
bool copyFile(const QString &source, const QString &dest)
{
    QFile sourceFile(source);
    if (!sourceFile.open(QIODevice::ReadOnly))
        return false;

    QFile destFile(dest);
    if (!destFile.open(QIODevice::WriteOnly))
        return false;

    destFile.write(sourceFile.readAll());

    return sourceFile.error() == QFile::NoError
           && destFile.error() == QFile::NoError;
}
```

在调用readAll()的那一行中，输入文件的所有内容都被读入到一个QByteArray中，然后将它传给write()函数以写到输出文件中。虽然获得QByteArray中的所有数据比逐项读取数据需要更多的内存，但是它也带来了一些方便。例如，在这之后可以使用qCompress()和qUncompress()函数来压缩和解压数据。与qCompress()和qUncompress()功能相当的另外一个不占内存的方法是使用Qt解决方案中的QtIOCompressor。QtIOCompressor压缩它写入的数据流，解压它要读入的数据流，而并不将整个文件存储在内存中。

其实还有比使用QDataStream更加合适的直接读取QIODevice的方案。除了一个字节都不能读(unread)的ungetChar()函数，QIODevice还提供了peek()函数，它能在不移动设备位置时返回下一个数据字节。这不仅对随机存取设备(诸如文件)有效，同时也对顺序存储设备(诸如网络套接)有效。另外，还有一个seek()函数用来设置设备的位置，它主要用以支持随机存取的设备。

二进制文件格式提供了数据存储最通用最紧凑的方式，而且QDataStream也使得存取二进制数据非常容易。除了本节中的例子，我们已经在第4章看到使用QDataStream来读写电子数据表格文件，而且第21章将再次看到QDataStream在这方面的应用，那里将使用QDataStream读写Windows指针文件。

12.2 读取和写入文本

虽然二进制文件格式比通常基于文本的格式更加紧凑,但是它们是机器语言,无法人工阅读或者编辑。在二进制文件格式无法适用的场合,可以使用文本格式来代替。Qt 提供了 QTextStream 类读写纯文本文件以及如 HTML、XML 和源代码等其他文本格式的文件。第 16 章将单独讨论如何处理 XML 文件。

QTextStream 考虑了 Unicode 编码与系统的本地编码或其他任意编码之间的转换问题,并且明确地处理了因使用不同操作系统而导致不同的行尾符之间的转换(在 Windows 操作系统上行尾符是"\r\n",UNIX 和 Mac OS X 操作系统上是"\n")。QTextStream 使用 16 位 QChar 类型基本数据单元。除了字符和字符串之外,QTextStream 还支持 C++ 基本数字类型,它可以进行基本数字类型和字符串之间的转换。例如,下面的代码将"Thomas M. Disch:334 \n"写到文件 sf-book.txt 中:

```
QFile file("sf-book.txt");
if (!file.open(QIODevice::WriteOnly)) {
    std::cerr << "Cannot open file for writing: "
              << qPrintable(file.errorString()) << std::endl;
    return;
}

QTextStream out(&file);
out << "Thomas M. Disch: " << 334 << endl;
```

写入文本数据非常容易,但读取文本却是一个挑战。因为文本数据(与使用 QDataStream 写入的二进制数据不同)从根本上说就是含糊而不确定的。让我们来看看下面的例子:

```
out << "Denmark" << "Norway";
```

如果 out 为 QTextStream,则实际上被写入的数据是字符串"DenmarkNorway"。我们真的不能期望下面的代码能够正确地读回数据:

```
in >> str1 >> str2;
```

实际上,str1 获得了整个词"DenmarkNorway",而 str2 什么也没有得到。使用 QDataStream 则不会发生这个问题,因为它在字符串数据前面保存了每个字符串的长度。

对于复杂的文件格式,成熟的解析器也许是必需的。解析器通常通过在 QChar 上使用 >> 来一个字符一个字符地读取数据,或者通过使用 QTextStream::readLine()来逐行读取数据。在本节的最后,将给出两个简单的例子,其中一个是逐行地读取输入文件,另一个则是一个字符一个字符地读取。对于一个处理全部文本的解析器,如果不考虑内存的使用大小或者已经知道所读文件很小的话,可以使用 QTextStream::readAll()一次读取整个文件。

在默认情况下,QTextStream 使用系统的本地编码(例如,在美国以及欧洲的大部分地区都可使用 ISO 8859-1 或 ISO 8859-15)进行读取与写入。当然这可以通过使用如下的 setCodec()而改变:

```
stream.setCodec("UTF-8");
```

本例中使用的 UTF-8 编码是一种与 ASCII 兼容的编码方式,它代表整个 Unicode 字符集。关于 Unicode 以及 QTextStrea 支持的更多编码信息,请参见第 18 章。

QTextStream 有各种各样的模仿 <iostream> 的选项。这些选项可以通过传递专门的对象,也称为流操作器,在流上设置以改变它的状态,或者是通过调用列在图 12.1 中的函数。下面的例子是在整型数 12 345 678 输出前设置了 showbase、uppercasedigits 以及 hex 选项,生成的输出文本为"0xBC614E":

```
out << showbase << uppercasedigits << hex << 12345678;
```

也可以通过成员函数来设置这些选项：

```
out.setNumberFlags(QTextStream::ShowBase
                   | QTextStream::UppercaseDigits);
out.setIntegerBase(16);
out << 12345678;
```

setIntegerBase(int)	
0	基于前缀自动检测（读取数据时）
2	二进制
8	八进制
10	十进制
16	十六进制

setNumberFlags(NumberFlags)	
ShowBase	如果基数是 2(“0b”)、8(“0”)或者 16(“0x”)，则显示前缀
ForceSign	在实数中总是显示符号
ForcePoint	在数字中总是显示十进制分隔符
UppercaseBase	使用基数前缀的大写版本(“0B”、“0X”)
UppercaseDigits	在十六进制数中使用大写字母

setRealNumberNotation(RealNumberNotation)	
FixedNotation Fixed	定点表示法(例如，“0.000123”)
ScientificNotation	科学计数表示法(例如；“1.234568e-04”)
SmartNotation	定点或者科学计数表示法，自动选用最紧凑的表示法

setRealNumberPrecision(int)
设置应该生成的最大位数(默认为 6 位)

setFieldWidth(int)
设置字段的最小尺寸(默认为 0)

setFieldAlignment(FieldAlignment)	
AlignLeft	填充字段的右边
AlignRight	填充字段的左边
AlignCenter	填充字段的两边
AlignAccountingStyle	在符号和数字之间填充
setPadChar(QChar)	设置用于填充字段的字符(默认为空格符)

setPadChar(QChar)
设置用于填充字段的字符(默认为空格符)

图 12.1　设置 QTextStream 选项的函数

与 QDataStream 相似，QTextStream 也是在 QIODevice 子类上运作的，它可以是 QFile、QTemporaryFile、QBuffer、QProcess、QTcpSocket 或者 QUdpSocket。此外，它还可以直接在 QString 上使用。例如：

```
QString str;
QTextStream(&str) << oct << 31 << " " << dec << 25 << endl;
```

这就使 str 中的内容为“37 25 \n”，因为十进制数的 31 表示八进制中的 37。这种情况下，就不必再为流设置编码，因为 QString 总是 Unicode 编码。

让我们看一个简单的基于文本文件格式的例子。在本书第一部分中所介绍的 Spreadsheet 应用程序中,使用了二进制数格式存储电子数据表格的数据。这种数据类由一个三元序列(行、列、公式)组成,每一非空单元都有同样的数据类型。以文本格式写入数据是显而易见的,下面是如何从一个修改的 Spreadsheet::writeFile()版本中提取数据的例子:

```
QTextStream out(&file);
for (int row = 0; row < RowCount; ++row) {
    for (int column = 0; column < ColumnCount; ++column) {
        QString str = formula(row, column);
        if (!str.isEmpty())
            out << row << " " << column << " " << str << endl;
    }
}
```

我们采用了一种简单的格式,每一行代表一个单元,而在行与列以及列与公式之间由空格分隔。公式可以包含空格,但是假设它不含"\n"("\n"是用来终止一行的符号)。现在看看相应的读取代码:

```
QTextStream in(&file);
while (!in.atEnd()) {
    QString line = in.readLine();
    QStringList fields = line.split(' ');
    if (fields.size() >= 3) {
        int row = fields.takeFirst().toInt();
        int column = fields.takeFirst().toInt();
        setFormula(row, column, fields.join(' '));
    }
}
```

我们一次读取一行电子数据表格的数据。readLine()函数将自动去掉行尾的"\n"。QString::split()返回一个字符串列表,只要哪里有给定的分隔符出现,它就会在哪里分隔字符串。例如,行"5 19 Total value"会变成含 4 个元素项的列表["5","19","Total","value"]。

如果在列表中包含三个以上的字段,则就准备开始提取数据了。QStringList::takeFirst()函数去掉列表中第一项并返回被去掉的项。我们使用它来提取行和列上的数据。我们并不执行任何的错误检查,如果读入了一个非整数的行或列的值,QString::toInt()就返回 0 值。当调用 setFormula()时,必须将剩下的字段连接成一个单独的字符串。

在第二个 QTextStream 例子中,将采用逐个字符的方法执行一个读取文本文件并输出相同文本的程序,其间将去除每行行尾的空格并将所有制表符用空格代替。这个程序的工作可以通过 tidyFile()函数来完成:

```
void tidyFile(QIODevice *inDevice, QIODevice *outDevice)
{
    QTextStream in(inDevice);
    QTextStream out(outDevice);

    const int TabSize = 8;
    int endlCount = 0;
    int spaceCount = 0;
    int column = 0;
    QChar ch;

    while (!in.atEnd()) {
        in >> ch;

        if (ch == '\n') {
            ++endlCount;
            spaceCount = 0;
            column = 0;
        } else if (ch == '\t') {
            int size = TabSize - (column % TabSize);
            spaceCount += size;
            column += size;
```

```
        } else if (ch == ' ') {
            ++spaceCount;
            ++column;
        } else {
            while (endlCount > 0) {
                out << endl;
                --endlCount;
                column = 0;
            }
            while (spaceCount > 0) {
                out << ' ';
                --spaceCount;
                ++column;
            }
            out << ch;
            ++column;
        }
    }
    out << endl;
}
```

我们基于传递给函数的 QIODevice 来创建一个输入和输出 QTextStream。除了当前的字符外，还支持三个状态跟踪变量：一个计算新行数(换行数)，一个计算空格数，一个标记当前所在行的当前列位置(用于将制表符转换为恰当数目的空格数)。

通过在 while 循环中对输入文件的每个字符进行迭代来完成整个解析过程。在某些地方的代码非常巧妙。例如，尽管将 TabSize 设置成了 8，但只是用正好足够的空格数来代替到下一制表符边界的制表符，而不是粗略地用 8 个空格来代替每一个制表符。如果获得一个换行符、一个制表符或者一个空格，仅仅只更新或修正状态数据。只有在获得另一种字符时才输出，并且在写出这种字符前，我们先写出未决的换行符和空格(以考虑空行和保留缩进)并更新状态数据。

```
int main()
{
    QFile inFile;
    QFile outFile;

    inFile.open(stdin, QFile::ReadOnly);
    outFile.open(stdout, QFile::WriteOnly);

    tidyFile(&inFile, &outFile);

    return 0;
}
```

这个例子不需要使用 QApplication 对象，因为我们只是使用 Qt 的工具类。有关这些类的列表，请查看 http://doc.trolltech.com/4.3/tools.html。我们已经假设这个程序是被用作过滤器的，例如：

```
tidy < cool.cpp > cooler.cpp
```

如果给出了文件名，这个程序可以很容易地扩展用来处理命令行上的文件名，另外还可以过滤 cin 到 cout。

因为这是一个控制台应用程序，它的 .pro 文件与我们所见过的用于图形用户界面应用程序的略微有些不同。

```
TEMPLATE     = app
QT           = core
CONFIG      += console
CONFIG      -= app_bundle
SOURCES      = tidy.cpp
```

我们只关联 QtCore,因为没有使用任何的图形用户界面功能。然后指定我们想在 Windows 系统下实现控制台输出,并且不希望在 Mac OS X 系统下同时存在许多应用程序。

对于读取和写入纯 ASCII 文件或者 ISO 8859-1 (Latin-1)文件,可以直接使用 QIODevice 的应用编程接口来代替使用 QTextStream。但这样做并不很明智,因为大多数应用程序都会在某处需要对其他编码的支持,而只有 QTextStream 对这些编码提供了完全连续的无缝支持。如果仍然想将文本直接写入 QIODevice,那么必须为 open()函数明确地指定 QIODevice::Text 标记,例如:

```
file.open(QIODevice::WriteOnly | QIODevice::Text);
```

当写入的时候,这个标记将通知 QIODevice 将"\n"字符转换为 Windows 下的"\r\n"序列。当读取的时候,这个标记将告知设备忽略所有平台环境中的"\n"字符。然后我们就可以假设:不管操作系统采用哪种行尾符转换,每一行结束都将使用一个"\n"换行符表示。

12.3 遍历目录

QDir 类提供了一种与平台无关的遍历目录并获得有关文件信息的方法。为了看看 QDir 是如何使用的,我们将编写一个小的控制台应用程序,它会计算一个特定目录以及这个目录下任意深度的子目录中所有图片所占用的空间。

应用程序的核心是 imageSpace()函数,它递归计算出给定目录中所有图片累加的大小总和:

```
qlonglong imageSpace(const QString &path)
{
    QDir dir(path);
    qlonglong size = 0;

    QStringList filters;
    foreach (QByteArray format, QImageReader::supportedImageFormats())
        filters += "*." + format;

    foreach (QString file, dir.entryList(filters, QDir::Files))
        size += QFileInfo(dir, file).size();

    foreach (QString subDir, dir.entryList(QDir::Dirs
                                           | QDir::NoDotAndDotDot))
        size += imageSpace(path + QDir::separator() + subDir);

    return size;
}
```

我们从使用给定的路径创建一个 QDir 对象开始,这个给定路径可能与当前目录或者绝对地址有关系。我们传递给 entryList()函数两个参数。第一个参数是文件名过滤器的一个列表,它们可以包含"*"和"?"这类的通配符。在这个实例中,将只选出含有 QImage 可以读取的文件格式。第二个参数指定所要的条目类型(普通文件、目录、驱动器等)。

我们遍历这个文件列表,把它们的大小累加起来。QFileInfo 类可以访问文件的属性,如文件的大小、权限、属主和时间戳,等等。

第二个 entryList()调用获得这个目录下所有子目录。我们遍历它们(不包括 . 和 ..)并且递归调用 imageSpace()以累计图片文件的大小。

为了创建每一个子目录的路径,我们把当前目录的路径和子目录名称组合起来,然后用斜线把它们分隔开。除了在 Windows 操作系统上认可"\"之外,QDir 在所有平台上都把"\"认作是目录分隔符。在把路径呈现给用户的时候,可以调用 QDir::convertSeparators(),这个静态函数把斜线转换为针对具体平台的正确的分隔符。

下面把 main()函数添加到小程序中:

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QStringList args = QCoreApplication::arguments();

    QString path = QDir::currentPath();
    if (args.count() > 1)
        path = args[1];

    std::cout << "Space used by images in " << qPrintable(path)
              << " and its subdirectories is "
              << (imageSpace(path) / 1024) << " KB" << std::endl;

    return 0;
}
```

我们使用 QDir::currentPath()将路径初始化为当前目录。作为备用选择,也可以使用 QDir::homePath()将它初始化为用户的主目录。如果用户在命令行中指定了一个路径,就使用它来替代前面的目录。最后,调用 imageSpace()函数来计算所有图片总共占用了多少空间。

QDir 类提供了其他一些与文件和目录相关的函数,如 entryInfoList()(它将返回 QFileInfo 对象的列表)、rename()、exists()、mkdir()和 rmdir()。QFile 类提供了一些方便的静态函数,包括 remove()和 exists()。同时,QFileSystemWatcher 可以通过发送 directoryChanged()和 fileChanged()信号,在目录或者文件发生任何改变时通知我们。

12.4　嵌入资源

到目前为止,本章已经讨论了如何在外部设备中存取数据,然而利用 Qt 还可以在应用程序的可执行文件中嵌入二进制数据或者文本。这可以通过使用 Qt 资源系统来实现。在其他的章节中,将使用资源文件将图片嵌入到可执行文件中,当然也可以嵌入其他种类的文件。与文件系统中的普通文件一样,嵌入的文件也可以通过 QFile 读取。

通过 Qt 资源编译器 rcc,可以将资源转换为 C++ 代码。还可以通过把下面一行代码加到 .pro 文件中来告诉 qmake 包括专门的规则以运行 rcc:

```
RESOURCES = myresourcefile.qrc
```

myresourcefile.qrc 文件是一个 XML 文件,它列出了所有嵌入到可执行文件中的文件。

假设我们正在编写一个保持联系细节信息的应用程序。考虑到用户使用的方便性,我们想在最后的可执行文件中嵌入国际拨号代码。如果文件在应用程序所建目录的 datafiles 目录下,那么资源文件将会如下所示:

```
<RCC>
<qresource>
    <file>datafiles/phone-codes.dat</file>
</qresource>
</RCC>
```

在应用程序中,资源是通过:/路径前缀识别的。在这个例子中,拨号代码文件的路径为:/datafiles/phone-codes.dat,它可以像其他任何文件一样通过 QFile 读取。

在可执行文件中的嵌入数据具有不易丢失的优点,而且也有利于创建真正独立的可执行文件(如果也采用了静态链接的话)。它的两个缺点是:第一,如果需要改变嵌入数据,则整个可执行文件都要跟着替换;第二,由于必须容纳被嵌入的数据,可执行文件本身将变得比较大。

Qt 资源系统所具备并提供的特征远不止本例中所介绍的这些,它还包括对文件名别名的支持和本地化的支持。在 http://doc.trolltech.com/4.3/resources.html 网站上有关于这些特性的文档。

12.5 进程间通信

QProcess 类允许我们执行外部程序并且和它们进行交互。这个类是异步工作的,且它在后台完成它的工作,这样用户界面就可以始终保持响应。当外部进程得到数据或者已经完成时,QProcess 会发出信号通知我们。

我们将查看一个小应用程序的代码,它为一个外部图片转换程序提供用户界面。对于这个实例,我们使用 ImageMagick 中的 convert 程序,它对于所有主要平台都可以免费得到。用户界面如图 12.2 所示。

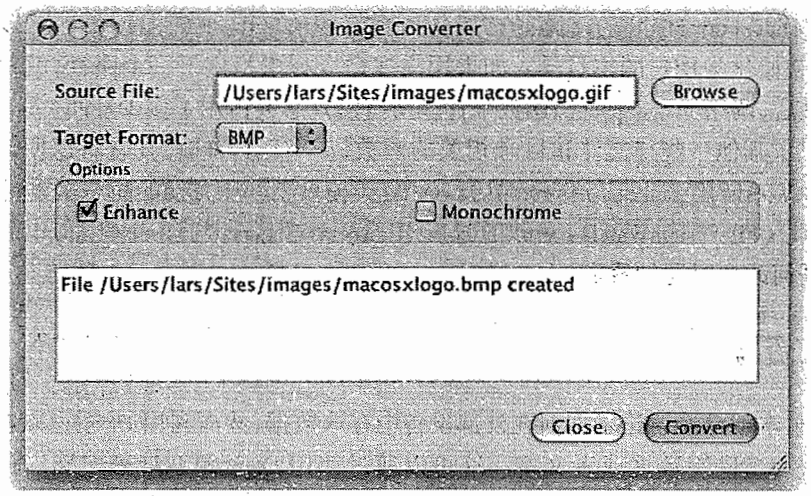


图 12.2 图片转换器应用程序

用户界面是在 Qt 设计师中创建的。.ui 文件就在本书的网站上中。这里,我们将主要查看由 uic 生成的 Ui::ConvertDialog 类派生的子类。开始的头文件如下:

```
#ifndef CONVERTDIALOG_H
#define CONVERTDIALOG_H

#include <QDialog>
#include <QProcess>
#include "ui_convertdialog.h"

class ConvertDialog : public QDialog, private Ui::ConvertDialog
{
    Q_OBJECT

public:
    ConvertDialog(QWidget *parent = 0);

private slots:
    void on_browseButton_clicked();
    void convertImage();
    void updateOutputTextEdit();
    void processFinished(int exitCode, QProcess::ExitStatus exitStatus);
    void processError(QProcess::ProcessError error);

private:
    QProcess process;
    QString targetFile;
};

#endif
```

头文件后面是 Qt 设计师窗体常见的子类模式。与我们见过的其他例子有些不同的是，这里已经私有继承了 Ui::ConvertDialog 类。这就防止了从外部窗体函数读取窗体控件。由于 Qt 设计师的自动关联机制，on_browseButton_clicked()和 on_convertButton_clicked()槽会被自动连接到 Browse 按钮的 clicked()信号。

```
ConvertDialog::ConvertDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);

    QPushButton *convertButton =
            buttonBox->button(QDialogButtonBox::Ok);
    convertButton->setText(tr("&Convert"));
    convertButton->setEnabled(false);

    connect(convertButton, SIGNAL(clicked()),
            this, SLOT(convertImage()));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
    connect(&process, SIGNAL(readyReadStandardError()),
            this, SLOT(updateOutputTextEdit()));
    connect(&process, SIGNAL(finished(int, QProcess::ExitStatus)),
            this, SLOT(processFinished(int, QProcess::ExitStatus)));
    connect(&process, SIGNAL(error(QProcess::ProcessError)),
            this, SLOT(processError(QProcess::ProcessError)));
}
```

setupUi()调用不仅创建并布置所有的窗体部件，还为 on_objectName_signalName()槽建立了信号-槽的连接。我们得到一个指向按钮框的 OK 按钮的指针，并给它赋上一个更合适的文本值。我们还使它失效，因为起初并没有需要转换的图像。同时，我们将它与 covertImage()槽相连。然后将按钮框的 rejected()信号(通过 Close 按钮发送)与对话框的 reject()槽相连接之后，手动将 QProcess 对象的三种信号与三种私有槽进行连接。只要外部进程在其 cerr 中有数据，就将在 updateOutput-TextEdit()中对其进行处理。

```
void ConvertDialog::on_browseButton_clicked()
{
    QString initialName = sourceFileEdit->text();
    if (initialName.isEmpty())
        initialName = QDir::homePath();
    QString fileName =
            QFileDialog::getOpenFileName(this, tr("Choose File"),
                                         initialName);
    fileName = QDir::toNativeSeparators(fileName);
    if (!fileName.isEmpty()) {
        sourceFileEdit->setText(fileName);
        buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
    }
}
```

通过 setupUi()，Browse 按钮的 clicked()信号被自动连接到 on_browseButton_clicked()槽。如果用户之前已经选择了一个文件，就用这个文件名来初始化对话框；否则，使用用户的主目录。

```
void ConvertDialog::convertImage()
{
    QString sourceFile = sourceFileEdit->text();
    targetFile = QFileInfo(sourceFile).path() + QDir::separator()
                 + QFileInfo(sourceFile).baseName() + "."
                 + targetFormatComboBox->currentText().toLower();
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(false);
    outputTextEdit->clear();

    QStringList args;
    if (enhanceCheckBox->isChecked())
```

```
        args << "-enhance";
    if (monochromeCheckBox->isChecked())
        args << "-monochrome";
    args << sourceFile << targetFile;

    process.start("convert", args);
}
```

当用户单击 Convert 按钮时,就复制源文件的名称并且根据目标文件格式改变它的扩展名。因为文件名对用户来说是可见的,所以使用针对特定平台的目录分隔符("/"或"\",如 QDir::separator()一样可用)代替手写的斜线。

然后,使 Convert 按钮失效,以避免用户意外启动多重转换,接着清空用于显示状态信息的文本编辑器。

为了启动外部进程,我们利用想运行的程序名称(convert)以及它所需的参数来调用 QProcess::start()。在这种情况下,如果用户勾选了正确的选项,就传递出由源文件名和目标文件名尾随的-enhance 和-monochrome 标记。convert 程序从文件扩展名来推断所要求的转换。

```
void ConvertDialog::updateOutputTextEdit()
{
    QByteArray newData = process.readAllStandardError();
    QString text = outputTextEdit->toPlainText()
                   + QString::fromLocal8Bit(newData);
    outputTextEdit->setPlainText(text);
}
```

只要外部进程向 cerr 写入,就会调用 updateOutputTextEdit()槽。我们读取错误文本并且把它添加到 QTextEdit 的现有文本中。

```
void ConvertDialog::processFinished(int exitCode,
                                    QProcess::ExitStatus exitStatus)
{
    if (exitStatus == QProcess::CrashExit) {
        outputTextEdit->append(tr("Conversion program crashed"));
    } else if (exitCode != 0) {
        outputTextEdit->append(tr("Conversion failed"));
    } else {
        outputTextEdit->append(tr("File %1 created").arg(targetFile));
    }
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
}
```

当进程已经完成时,会将结果通知用户然后激活 Convert 按钮:

```
void ConvertDialog::processError(QProcess::ProcessError error)
{
    if (error == QProcess::FailedToStart) {
        outputTextEdit->append(tr("Conversion program not found"));
        buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
    }
}
```

如果进程不能开启,QProcess 就会发出 error()而不是 finished()。我们将报告所有的错误并激活 Click 按钮。

这个例子异步地执行了文件的转换,也就是说,我们告知 QProcess 去运行 convert 程序并立即将控制权返回给应用程序。当进程在后台运行时,这可以让用户界面始终保持响应。但在某些情况下,在自己的应用程序进一步执行前,必须先完成外部进程。这时需要同步操作 QProcess。

利用用户首选的文本编辑器,支持纯文本编辑的应用程序,是一个需要用到同步状态的常见的例子。采用 QProcess 就可以直接实现。例如,假设在 QTextEdit 中有纯文本,并且提供用户可以点击的连接到 edit()槽的 Edit 按钮。

```
void ExternalEditor::edit()
{
    QTemporaryFile outFile;
    if (!outFile.open())
        return;

    QString fileName = outFile.fileName();
    QTextStream out(&outFile);
    out << textEdit->toPlainText();
    outFile.close();

    QProcess::execute(editor, QStringList() << options << fileName);

    QFile inFile(fileName);
    if (!inFile.open(QIODevice::ReadOnly))
        return;

    QTextStream in(&inFile);
    textEdit->setPlainText(in.readAll());
}
```

我们使用 QTemporaryFile 创建一个具有唯一文件名的空文件。不给 QTemporaryFile::open()指定任何参数,因为它在读/写模式下默认是打开的。我们给临时文件写入编辑文本的内容,然后关闭文件,因为一些文本编辑器不能在已打开的文件中运行。

QProcess::execute()静态函数运行外部进程并当该外部进程完成时停止。editor 参数是具有编辑器可执行文件名的 QString(例如,"gvim")。options 参数则为一个 QStringList(如果正使用 gvim,则包含一项,"-f")。

在用户关闭文本编辑器后,进程结束且 execute()函数的调用也将返回。然后,打开临时文件并将临时文件的内容读到 QTextEdit 中。当对象超出作用范围时,QTemporaryFile 将自动删除临时文件。

当 QProcess 同步使用时,并不需要信号-槽之间的连接。如果需要比 execute()静态函数所提供的更好的控制,可以使用另一种方式。这就涉及到要创建 QProcess 对象并对它调用 start(),然后通过调用 QProcess::waitForStarted()强制使它停止。如果成功,再调用 QProcess::waitForFinished()。可以查阅 QProcess 的参考文档,看看使用这种方法的例子。

在本节中,使用了 QProcess 让我们有权使用先前已经存在的功能。使用已有的应用程序可以节省开发时间,同时也使我们远离了那些与主应用程序目的不太相关的细节问题。使用先前已经存在的功能的另一种方法是连接到一个提供这些功能的数据库。但若没有一个合适的数据库时,采用 QProcess 包装一个控制台应用程序也是很有用的。

QProcess 的另一个用处是:还可以启动其他的用户图形界面应用程序。然而,如果目标是建立应用程序之间的关联,而不是简单地从一个应用程序中调用运行另一个,则最好采用 Qt 的网络类或其在 Windows 下的 ActiveQt 扩展程序,让应用程序之间能够更好地实现直接通信。而如果想启动用户喜欢的网页浏览器或者电子邮件客户端程序,仅仅只需要调用 QDesktopServices::openUrl()。

第13章 数 据 库

QtSql 模块提供了与平台以及数据库种类无关的访问 SQL 数据库的接口。这个接口由利用 Qt 的模型/视图结构将数据库与用户界面集成的一套类来支持。本章内容是以假设读者已经熟悉掌握了第 10 章所讲到的 Qt 中的模型/视图类为前提的。

QSqlDatabase 对象表征了数据库的关联。Qt 使用驱动程序与各种数据库的应用编程接口进行通信。Qt 的桌面版(Desktop Edition)包括如下一些驱动程序:

驱动程序	数据库
QDB2	IBM DB2 7.1 版以及更新的版本
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	甲骨文公司(Oracle Call Interface)
QODBC	ODBC(包括微软公司的 SQL 服务器)
QPSQL	PostgreSQL 的 7.3 版以及更高级的版本
QSQLITE	SQLite 第 3 版
QSQLITE2	SQLite 第 2 版
QTDS	Sybase 自适应服务器

由于授权许可的限制,Qt 的开源版本无法提供所有的驱动程序。当配置 Qt 时,既可以选择 Qt 本身就包含的 SQL 驱动程序,也可以以插件的形式建立驱动程序。公共领域中不断发展的 SQLite 数据库将向 Qt 提供支持①。

对于那些习惯了 SQL 语法的用户,QSqlQuery 类提供了一种直接执行任意的 SQL 语句并处理其结果的方式。对于那些喜欢更高级、更友好的数据库界面以避免 SQL 语法的用户,QSqlTableModel 和 QSqlRelationalTableModel 提供了合适的抽象。这些类以与 Qt 其他模型类(见第 10 章)相同的方式来表示一个 SQL 表。它们可以被单独用来遍历和编辑程序代码中的数据,也可以添加最终用户能够查看并修改数据的视图。

Qt 使对常见数据库特性的编程变得简单易懂,例如对主从数据库(master-detail)和下钻型(drill-down)数据库的编程;另外,Qt 也让利用窗体或图形用户界面中的表查看数据库表的过程变得简单易行,例如本章中将要演示的几个实例。

13.1 连接和查询

为了执行 SQL 查询,首先必须建立与数据库的连接。通常情况下,是在应用程序开始时所调用的一个单独的函数中建立数据库连接。例如:

```
bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
```

① 在构建 Qt 时,必须启动 SQL 的支持。例如,通过在 configure 脚本中传递-qt-sql-sqlite 命令行选项,或者在 Qt 安装器中设置适当的选项,可以构建出对 SQLite 提供内置支持的 Qt。

```
    db.setHostName("mozart.konkordia.edu");
    db.setDatabaseName("musicdb");
    db.setUserName("gbatstone");
    db.setPassword("T17aV44");
    if (!db.open()) {
        QMessageBox::critical(0, QObject::tr("Database Error"),
                              db.lastError().text());
        return false;
    }
    return true;
}
```

首先,调用 QSqlDatabase::addDatabase()来创建 QSqlDatabase 对象。addDatabase()的第一个参数指定了 Qt 必须使用哪一个数据库驱动程序来访问这个数据库,这里使用的是 MySQL。

接下来,设置数据库的主机名、数据库名、用户名和密码,并且打开这个连接。如果 open()操作失败,将显示出错信息。

通常情况下,是在 main()中调用 createConnection()的:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!createConnection())
        return 1;
    ...
    return app.exec();
}
```

一旦连接建立,就可以使用 QSqlQuery 执行底层数据库支持的任何 SQL 语句了。例如,以下是如何执行 SELECT 语句的代码:

```
QSqlQuery query;
query.exec("SELECT title, year FROM cd WHERE year >= 1998");
```

在 exec()调用之后,可以遍历查询的结果集:

```
while (query.next()) {
    QString title = query.value(0).toString();
    int year = query.value(1).toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

只要调用 next()一次,就可以把这个 QSqlQuery 定位到结果集中的第一条记录。随后的 next()调用,每次都会把记录指针前移一条记录,直到到达结尾时 next()才返回 false。如果结果集为空(或查询失败),那么 next()的第一次调用将返回 false。

value()函数把字段值作为 QVariant 返回。字段是按照 SELECT 语句中给定的顺序从编号 0 开始的。这个 QVariant 类可以保存许多 C++ 和 Qt 数据类型,包括 int 和 QString。可存储于数据库中的不同数据类型都可以映射为相应的 C++ 和 Qt 类型并且存储到 QVariant 中。例如,VARCHAR 代表着 QString,而 DATETIME 代表着 QDateTime。

QSqlQuery 提供了一些可以遍历结果集的函数:first()、last()、previous()和 seek()。这些函数都很方便,不过对于某些数据库,它们可能会比 next()更慢或者更加耗费内存。在操作一个大数据集时,为了便于优化,可以在调用 exec()之前调用 QSqlQuery::setForwardOnly(true),然后只使用 next()遍历结果集。

之前,我们把这个 SQL 查询指定为 QSqlQuery::exec()的参数,然而还可以直接把它传递给构造函数,构造函数会立即执行 SQL 查询:

```
QSqlQuery query("SELECT title, year FROM cd WHERE year >= 1998");
```

可以通过对查询调用 isActive()来检查是否有错误发生:

```
if (!query.isActive())
    QMessageBox::warning(this, tr("Database Error"),
                         query.lastError().text());
```

如果没有错误发生,则查询会变成"激活"状态,然后就可以使用 next()来遍历结果集。

执行 INSERT 的代码与执行 SELECT 的一样容易:

```
QSqlQuery query("INSERT INTO cd (id, artistid, title, year) "
                "VALUES (203, 102, 'Living in America', 2002)");
```

在这之后,numRowsAffected()返回受 SQL 语句影响的行数(如果发生错误,就返回 -1)。

如果需要插入多条记录,或者想避免将数值转换成为字符串(并且正确地转义它们),可以使用 prepare()来指定一个包含占位符的查询,然后赋值绑定想插入的数值。Qt 对所有数据库都支持 Oracle 风格和 ODBC 风格的占位符语法,如果它们可用,就使用本地支持;如果不可用,就模拟它的功能。以下是使用 Oracle 风格语法及命名占位符的实例:

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (:id, :artistid, :title, :year)");
query.bindValue(":id", 203);
query.bindValue(":artistid", 102);
query.bindValue(":title", "Living in America");
query.bindValue(":year", 2002);
query.exec();
```

以下是一个使用 ODBC 风格位置占位符的相同实例:

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (?, ?, ?, ?)");
query.addBindValue(203);
query.addBindValue(102);
query.addBindValue("Living in America");
query.addBindValue(2002);
query.exec();
```

在 exec()调用之后,可以调用 bindValue()或者 addBindValue()来赋值绑定新值,然后再次调用 exec()并利用这些新值执行查询。

占位符通常用于指定二进制数据或者包含非 ASCII 码或者非 Latin 字符的字符串。在底层,Qt 对支持 Unicode 的数据库使用 Unicode 编码,而对于不支持 Unicode 的,Qt 会明确地把字符串转换为合适的编码方式。

如果数据库中 SQL 事务处理可用的话,Qt 就支持它。为了开始事务处理,需对代表数据库连接的 QSqlDatabase 对象调用 transaction()。为了结束事务处理,可以调用 commit()或者 rollback()。例如,以下是如何查询外键并且在事务处理中执行 INSERT 语句的代码:

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROM artist WHERE name = 'Gluecifer'");
if (query.next()) {
    int artistId = query.value(0).toInt();
    query.exec("INSERT INTO cd (id, artistid, title, year) "
               "VALUES (201, " + QString::number(artistId)
               + ", 'Riding the Tiger', 1997)");
}
QSqlDatabase::database().commit();
```

QSqlDatabase::database()函数返回一个表示在 createConnection()中创建的连接的 QSqlDatabase 对象。如果事务处理不能启动,QSqlDatabase::transaction()就返回 false 值。一些数据库不支持事

务处理。对于这类数据库，transaction()、commit()和 rsollback()几个函数就什么也不做。可以使用 hasFeature()对数据库相关的 QSqlDriver 进行测试，看看这个数据库是不是支持事务处理。

```
QSqlDriver *driver = QSqlDatabase::database().driver();
if (driver->hasFeature(QSqlDriver::Transactions))
    ...
```

还可以测试其他的一些数据库特征，包括数据库是否支持 BLOB(二进制大对象)、Unicode 以及经过处理的查询。

使用 QSqlDriver::handle()和 QSqlResult::handle()函数，还可以读取低级数据库驱动程序句柄和查询结果集的低级句柄。但如果不清楚其使用目的与细节的话，这两个函数的使用非常容易出错。读者可以看看有关它们的一些资料中的实例及其风险说明。

目前为止，在我们研究的实例中，都是假设应用程序所使用的是单一的数据库连接。如果想创建多个连接，可以把数据库名作为第二个参数传递给 addDatabase()。例如：

```
QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL", "OTHER");
db.setHostName("saturn.mcmanamy.edu");
db.setDatabaseName("starsdb");
db.setUserName("hilbert");
db.setPassword("ixtapa7");
```

然后，可以通过把数据库名传递给 QSqlDatabase::database()得到指向 QSqlDatabase 对象的指针：

```
QSqlDatabase db = QSqlDatabase::database("OTHER");
```

为了使用其他连接执行查询，我们把 QSqlDatabase 对象传递给 QSqlQuery 的构造函数：

```
QSqlQuery query(db);
query.exec("SELECT id FROM artist WHERE name = 'Mando Diao'");
```

如果想一次执行多个事务处理，多重连接是很有用的，因为每个连接只能处理一个有效的事务处理。当使用多个数据库连接时，还可以有一个命名的连接，而且如果没有具体指定的话，QSqlQuery 就会使用这个未命名的连接。

除了 QSqlQuery 之外，Qt 提供了 QSqlTableModel 类作为一个高级界面接口，让我们不必使用原始的 SQL 语句来执行大多数常用的 SQL 操作(如 SELECT、INSERT、UPDATE 和 DELETE)。这个类可以用来独立处理数据库而不涉及任何的图形用户界面，它也可以用作 QListView 或 QTableView 的数据源。

以下是使用 QSqlTableModel 执行 SELECT 操作的实例：

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("year >= 1998");
model.select();
```

这等价于如下的查询：

```
SELECT * FROM cd WHERE year >= 1998
```

利用 QSqlTableModel::record()获得某一给定的记录，或者利用 value()读取单独的字段，我们可以遍历这个结果集：

```
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value("title").toString();
    int year = record.value("year").toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

QSqlRecord::value()函数既可以接收字段名也可以接收字段索引。在对大数据集进行操作时，建议利用索引来指定字段。例如：

```
int titleIndex = model.record().indexOf("title");
int yearIndex = model.record().indexOf("year");
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value(titleIndex).toString();
    int year = record.value(yearIndex).toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

为了在数据库表中插入记录,可调用 insertRow()来创建一个新的空行(记录),然后使用 setData 设置每一个列(字段)的值:

```
QSqlTableModel model;
model.setTable("cd");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 0), 113);
model.setData(model.index(row, 1), "Shanghai My Heart");
model.setData(model.index(row, 2), 224);
model.setData(model.index(row, 3), 2003);
model.submitAll();
```

在调用 submitAll()之后,记录可能会被移动到一个不同的行位置,这取决于表是如何排序的。如果插入失败,submitAll()调用将返回 false。

SQL 模型与标准模型之间最大的区别在于:对于 SQL 模型,必须调用 submitAll()以将发生的更改写入数据库。

为了更新某一记录,首先必须把 QSqlTableModel 定位到要修改的记录上[例如使用 select()]。然后提取这条记录,更新想改变的字段并将更改过的数据回写到数据库中:

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    QSqlRecord record = model.record(0);
    record.setValue("title", "Melody A.M.");
    record.setValue("year", record.value("year").toInt() + 1);
    model.setRecord(0, record);
    model.submitAll();
}
```

如果有一条记录与指定的过滤器相匹配,就利用 QSqlTableModel::record()获得这条记录。并用修改后的记录复写原始的记录。

正如对非 SQL 模型所作的处理一样,也可以使用 setData()来执行更新。我们获得的模型索引都是针对给定的行与列的:

```
model.select();
if (model.rowCount() == 1) {
    model.setData(model.index(0, 1), "Melody A.M.");
    model.setData(model.index(0, 3),
                  model.data(model.index(0, 3)).toInt() + 1);
    model.submitAll();
}
```

删除记录与更新记录的过程很相似:

```
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    model.removeRows(0, 1);
    model.submitAll();
}
```

removeRows()调用删除了第一条记录的行号以及记录号。下面的实例则删除了所有与过滤器匹配的记录：

```
model.setTable("cd");
model.setFilter("year < 1990");
model.select();
if (model.rowCount() > 0) {
    model.removeRows(0, model.rowCount());
    model.submitAll();
}
```

QSqlQuery 和 QSqlTableModel 这两个类提供了 Qt 和 SQL 数据库之间的接口。利用这些类，可以创建显示用户数据以及让用户插入、更新和删除记录的表单。

对于使用 SQL 类的应用程序，需要将如下的命令行添加到其 .pro 文件中：

```
QT += sql
```

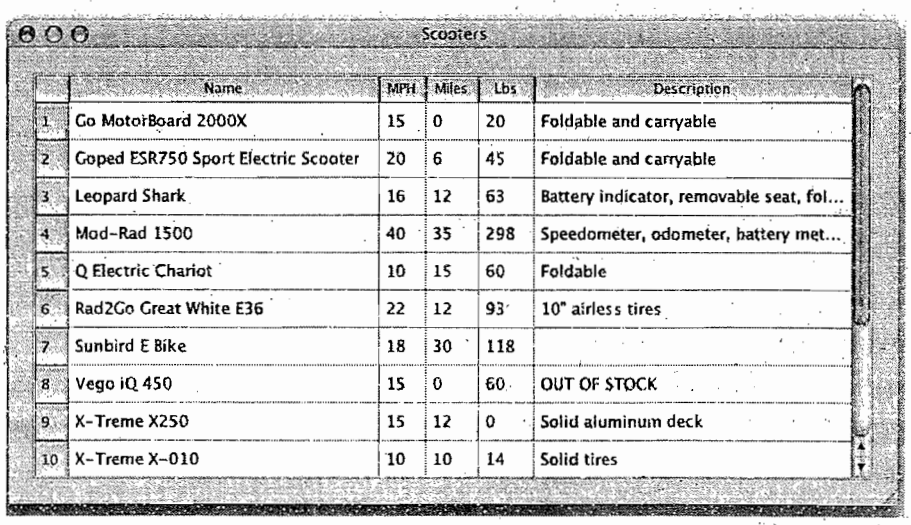
这将确保应用程序可以连接到 QtSql 库。

13.2　查看表

在前一节中，我们看到了如何使用 QSqlQuery 和 QSqlTableModel 与数据库进行交互。在本节中，将看看如何在 QTableView 窗口部件中显示 QSqlTableModel。

如图 13.1 所示的 Scooters 应用程序，给出了踏板车(scooter)的型号表。该实例基于单一的 scooter 表，其定义如下：

```
CREATE TABLE scooter (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL,
    maxspeed INTEGER NOT NULL,
    maxrange INTEGER NOT NULL,
    weight INTEGER NOT NULL,
    description VARCHAR(80) NOT NULL);
```



	Name	MPH	Miles	Lbs	Description
1	Go MotorBoard 2000X	15	0	20	Foldable and carryable
2	Goped ESR750 Sport Electric Scooter	20	6	45	Foldable and carryable
3	Leopard Shark	16	12	63	Battery indicator, removable seat, fol...
4	Mod-Rad 1500	40	35	298	Speedometer, odometer, battery met...
5	Q Electric Chariot	10	15	60	Foldable
6	Rad2Go Great White E36	22	12	93	10" airless tires
7	Sunbird E Bike	18	30	118	
8	Vego iQ 450	15	0	60	OUT OF STOCK
9	X-Treme X250	15	12	0	Solid aluminum deck
10	X-Treme X-010	10	10	14	Solid tires

图 13.1　Scooters 应用程序

在这种采用 SQLite 的情况下，id 字段的值都是由数据库自动生成的，而其他类型的数据库可能会使用不同的语法来完成。

为了便于维护,我们利用枚举类型变量 enum 为表的列索引号给出有具体含义的命名:

```
enum {
    Scooter_Id = 0,
    Scooter_Name = 1,
    Scooter_MaxSpeed = 2,
    Scooter_MaxRange = 3,
    Scooter_Weight = 4,
    Scooter_Description = 5
};
```

如下给出了建立 QSqlTableModel 以显示 scooter 表所需的代码:

```
model = new QSqlTableModel(this);
model->setTable("scooter");
model->setSort(Scooter_Name, Qt::AscendingOrder);
model->setHeaderData(Scooter_Name, Qt::Horizontal, tr("Name"));
model->setHeaderData(Scooter_MaxSpeed, Qt::Horizontal, tr("MPH"));
model->setHeaderData(Scooter_MaxRange, Qt::Horizontal, tr("Miles"));
model->setHeaderData(Scooter_Weight, Qt::Horizontal, tr("Lbs"));
model->setHeaderData(Scooter_Description, Qt::Horizontal,
                     tr("Description"));
model->select();
```

创建模型的过程与上节中所述的过程相似。不同之处在于,这里使用了我们自己的列标题。如果不这样做,则列标题将采用原始的字段名。我们还使用 setSort()指定了一个排序,它在后台运行的程序中通过 ORDER BY 子句实现。

我们已经创建了模型并利用 select()将其以数据组装,还可以创建一个视图来显示它:

```
view = new QTableView;
view->setModel(model);
view->setSelectionMode(QAbstractItemView::SingleSelection);
view->setSelectionBehavior(QAbstractItemView::SelectRows);
view->setColumnHidden(Scooter_Id, true);
view->resizeColumnsToContents();
view->setEditTriggers(QAbstractItemView::NoEditTriggers);

QHeaderView *header = view->horizontalHeader();
header->setStretchLastSection(true);
```

在第 10 章中,我们看到了如何使用 QTableView 在一个表中显示来自 QAbstractItemModel 的数据。因为 QSqlTableModel(间接)派生自 QAbstractItemModel,它可以很容易地作为 QTableView 数据源使用。setModel()调用是连接视图和模型唯一必需的调用,其余的代码仅用来定制表以使其具有更加友好的用户界面。

选择模式指定了用户可以选用的所有可能模式,这里我们使单独的单元格(字段)可选。被选中的单元格通常用一个带点的轮廓线包围。选择状态指定了被选项如何在考虑整行的情况下实现可视的效果。被选项通常由不同的背景颜色来表示。这里选择了隐藏 ID 列,因为对于用户来说,ID 并无意义。此外,还设置 NoEditTriggers 以使表视图具有只读属性。

显示只读表的另一方式是使用 QSqlTableModel 的基类——QSqlQueryModel。该类提供 setQuery()函数,因此它可设置复杂的 SQL 查询以提供含一个或多个表的专门视图——例如,使用 SQL 表连接算法(join)。

与 Scooters 应用程序的数据库不同,大多说数据库都存在着大量的表与外键关联(在关系数据库中,外键表示两表间的引用约束)。Qt 提供了 QSqlRelationalTableModel,它是一个可以利用外键来显示和编辑表的 QSqlTableModel 子类。除了可以为每一个外键将 QSqlTableModel 添加到模型以外,QSqlRelationalTableModel 与 QSqlTableModel 的功能非常相似。在许多情况下,外键有一个 ID 字段和一个命名字段。虽然在后台程序中相应的 ID 字段才是真正被使用的字段,但利用 QSqlRelational-

TableModel，可以确保用户能看到和更改外键的命名字段。为了让其有效运作，必须对用于显示模型的视图设置一个 QSqlRelationalDelegate 类（或者是一个用户自定义的子类）。

在随后的两节中，将看到如何实现显示功能及外键更改，还将在本章的最后一节给出更多关于 QTableViews 内容。

13.3　使用窗体编辑记录

在本节中，我们将看到如何创建一次只显示一条记录的对话窗体。这个对话框可以用于增加、编辑、删除单独的记录，也可以遍历表中所有的记录。

我们将通过 Staff Manager 应用程序来阐明这些概念。该应用程序记录了雇员所属的部门、部门所处的位置以及诸如雇员内部电话分机号码等一些基本信息。应用程序使用了如下三个表：

```
CREATE TABLE location (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL));

CREATE TABLE department (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL,
    locationid INTEGER NOT NULL,
    FOREIGN KEY (locationid) REFERENCES location));

CREATE TABLE employee (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL,
    departmentid INTEGER NOT NULL,
    extension INTEGER NOT NULL,
    email VARCHAR(40) NOT NULL,
    startdate DATE NOT NULL,
    FOREIGN KEY (departmentid) REFERENCES department));
```

表以及表之间的关系示意见图 13.2。在每一地点可以有许多部门，而每一部门可以拥有许多雇员。指定外键的语法主要针对 SQLite3，但可能会随数据库的不同而有所变化。

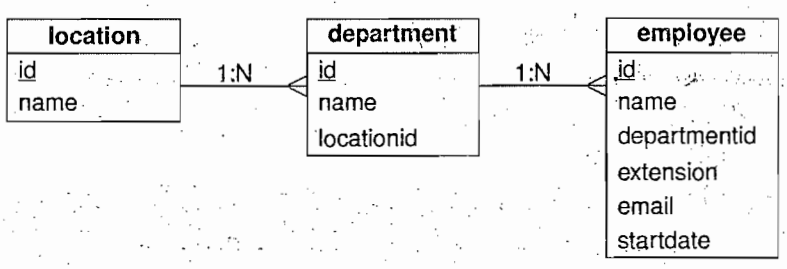


图 13.2　Staff Manager 应用程序中的表

本节将重点关注编辑雇员信息用的对话框 EmployeeForm。下一节讨论的是 MainForm，它提供了部门和雇员的主从关系视图。

EmployeeForm 类提供了一个从主窗体雇员概要信息到某一雇员的具体细节信息的下钻型视图。当调用该类时，如果给出了有效的雇员 ID，窗体将显示指定的雇员信息；否则显示第一个雇员的信息（窗体如图 13.3 所示）。用户可以浏览查看所有雇员的信息，编辑或者删除现有雇员的信息，同时还可以添加新雇员的信息。

employeeform.h 文件中已经提供了如下的 enum 枚举类型变量，为表的列索引号给出有具体含义的名称：

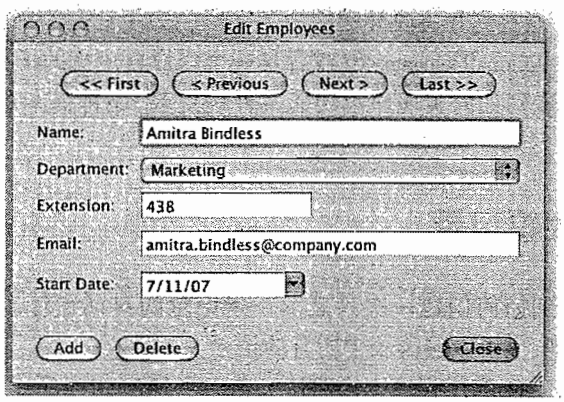


图 13.3 雇员信息对话框

```
enum {
    Employee_Id = 0,
    Employee_Name = 1,
    Employee_DepartmentId = 2,
    Employee_Extension = 3,
    Employee_Email = 4,
    Employee_StartDate = 5
};
```

头文件的其余部分则定义 EmployeeForm 类:

```
class EmployeeForm : public QDialog
{
    Q_OBJECT
public:
    EmployeeForm(int id, QWidget *parent = 0);

    void done(int result);

private slots:
    void addEmployee();
    void deleteEmployee();

private:
    QSqlRelationalTableModel *tableModel;
    QDataWidgetMapper *mapper;
    QLabel *nameLabel;
    ...
    QDialogButtonBox *buttonBox;
};
```

为了访问数据库,我们使用 QSqlRelationalTableModel 而不是普通的 QSqlTableModel,因为需要解析表之间的外键关系。QSqlRelationalTableModel 是允许将某一窗体中的窗口部件映射到数据模型中对应的列的类。

窗体的构造函数非常长,所以将其分为几个部分,逐段进行分析。至于没有太多实质性用途的版面布局代码,我们就不去分析了。

```
EmployeeForm::EmployeeForm(int id, QWidget *parent)
    : QDialog(parent)
{
    nameEdit = new QLineEdit;

    nameLabel = new QLabel(tr("Na&me:"));
    nameLabel->setBuddy(nameEdit);

    departmentComboBox = new QComboBox;

    departmentLabel = new QLabel(tr("Depar&tment:"));
    departmentLabel->setBuddy(departmentComboBox);
```

```
extensionLineEdit = new QLineEdit;
extensionLineEdit->setValidator(new QIntValidator(0, 99999, this));

extensionLabel = new QLabel(tr("E&xtension:"));
extensionLabel->setBuddy(extensionLineEdit);

emailEdit = new QLineEdit;

emailLabel = new QLabel(tr("&Email:"));
emailLabel->setBuddy(emailEdit);

startDateEdit = new QDateEdit;
startDateEdit->setCalendarPopup(true);
QDate today = QDate::currentDate();
startDateEdit->setDateRange(today.addDays(-90), today.addDays(90));

startDateLabel = new QLabel(tr("&Start Date:"));
startDateLabel->setBuddy(startDateEdit);
```

首先为每个字段创建一个可编辑的窗口部件。还创建了对应的标签,放在每个可编辑的窗口部件旁边,用以识别相应的字段。

我们使用 QIntValidator 确保 Extension 行编辑器仅接受有效的分机号。这样的话,所有的号码都将在 0 ~ 99 999 的范围之内。我们也为 Start Date 编辑器设置了数值范围,同时设置编辑器以提供一个弹出式的日历。我们并不直接组装组合框,稍后将给它一个能自我组装的模型。

```
firstButton = new QPushButton(tr("<< &First"));
previousButton = new QPushButton(tr("< &Previous"));
nextButton = new QPushButton(tr("&Next >"));
lastButton = new QPushButton(tr("&Last >>"));

addButton = new QPushButton(tr("&Add"));
deleteButton = new QPushButton(tr("&Delete"));
closeButton = new QPushButton(tr("&Close"));

buttonBox = new QDialogButtonBox;
buttonBox->addButton(addButton, QDialogButtonBox::ActionRole);
buttonBox->addButton(deleteButton, QDialogButtonBox::ActionRole);
buttonBox->addButton(closeButton, QDialogButtonBox::AcceptRole);
```

我们创建浏览导航按钮(<<First、<Previous、Next> 和 Last>>),这些按钮都集中在对话框的顶部。然后,创建其他的按钮(诸如 Add、Delete 和 Close),并将它们放在位于对话框底部的 QDialogButtonBox 中。创建版面布局的代码很简单,这里不再赘述。

这样,我们就完成了组成用户接口界面的窗口部件,因此现在可以将注意力转移到如何实现其内在的功能上。

```
tableModel = new QSqlRelationalTableModel(this);
tableModel->setTable("employee");
tableModel->setRelation(Employee_DepartmentId,
                        QSqlRelation("department", "id", "name"));
tableModel->setSort(Employee_Name, Qt::AscendingOrder);
tableModel->select();

QSqlTableModel *relationModel =
        tableModel->relationModel(Employee_DepartmentId);
departmentComboBox->setModel(relationModel);
departmentComboBox->setModelColumn(
        relationModel->fieldIndex("name"));
```

模型的构建与设置过程都与之前介绍的 QSqlTableModel 的设立过程基本一致。但是这次使用 QSqlRelationalTableModel 并设立一个外键关联。setRelation()函数获得一个外键字段索引及 QSqlRelation 索引。QSqlRelation 构造函数则用表名(外键关系对应的表)、外键字段名以及要显示的字段名来表示外键字段值。

QComboBox 与 QListWidget 很相似,因为它有一个内部模型去保存它的数据条目项。我们可以

用自己建的模型代替那个模型,这里需要做的,就是给出 QSqlRelationalTableModel 使用的关系模型。这个关系模型有两列,所以必须指出组合框应该显示的是哪一列。当调用 setRelation()时,就建立关系模型,所以我们不知道 name 列的标题索引。由于这个原因,我们使用 fieldIndex()函数与字段名得到正确的标题索引,以使组合框显示部门名。由于 QSqlRelationalTableModel 的使用,组合框将显示部门名而不是部门的 ID 号。

```
mapper = new QDataWidgetMapper(this);
mapper->setSubmitPolicy(QDataWidgetMapper::AutoSubmit);
mapper->setModel(tableModel);
mapper->setItemDelegate(new QSqlRelationalDelegate(this));
mapper->addMapping(nameEdit, Employee_Name);
mapper->addMapping(departmentComboBox, Employee_DepartmentId);
mapper->addMapping(extensionLineEdit, Employee_Extension);
mapper->addMapping(emailEdit, Employee_Email);
mapper->addMapping(startDateEdit, Employee_StartDate);
```

QDataWidgetMapper 将一个数据库记录字段反映到其映射的窗口部件中,同时将窗口部件中所做出的更改反映回数据库。我们既可以自己负责提交这些更改,也可以让映射器自动完成这个工作,这里选择自动选项(QDataWidgetMapper::AutoSubmit)。

让模型有效工作的映射器必须给定,对于含有外键的模型,还需要给它一个 QSqlRelationalDelegate 委托基类。这个委托基类可以确保用户看到的是 QSqlRelation 显示栏的值,而不是原始的 ID 号,同时它也保证在用户开始编辑时,组合框会显示值,而映射器则实际上将相应的索引值(外键)写回到数据库中。

对于外键引用约束含有大量记录的表的情况,最好创建自己的委托基类并与搜索性能一起使用以显示"列表值"窗体,而不是依靠 QSqlRelationalTableModel 的默认组合框。

一旦模型和委托基类都设置好,就可在窗体的窗口部件及其相应的字段索引间添加映射关系。组合框则被当作其他的一般窗口部件来处理,因为所有处理外键的工作都将用已经设置好的关系模型来操作。

```
if (id != -1) {
    for (int row = 0; row < tableModel->rowCount(); ++row) {
        QSqlRecord record = tableModel->record(row);
        if (record.value(Employee_Id).toInt() == id) {
            mapper->setCurrentIndex(row);
            break;
        }
    }
} else {
    mapper->toFirst();
}
```

如果对话框通过一个有效的雇员 ID 号而调用,我们就利用这个 ID 查找这个雇员的记录,并让它成为映射器的当前记录。否则,我们只浏览到第一个雇员记录。或者,在另一种情况下,记录的数据将被反映到其映射的窗口部件中。

```
    connect(firstButton, SIGNAL(clicked()), mapper, SLOT(toFirst()));
    connect(previousButton, SIGNAL(clicked()),
            mapper, SLOT(toPrevious()));
    connect(nextButton, SIGNAL(clicked()), mapper, SLOT(toNext()));
    connect(lastButton, SIGNAL(clicked()), mapper, SLOT(toLast()));
    connect(addButton, SIGNAL(clicked()), this, SLOT(addEmployee()));
    connect(deleteButton, SIGNAL(clicked()),
            this, SLOT(deleteEmployee()));
    connect(closeButton, SIGNAL(clicked()), this, SLOT(accept()));
    ...
}
```

导航浏览按钮都是直接连接到相应的映射器槽的。(如果使用手动提交的策略,则需要实现自定义的这些槽,在这些槽中提交当前记录,然后执行导航浏览以避免所做的修改丢失。)数据窗口部件映射器允许进行编辑和导航浏览。若要增加和删除记录,可使用底层的模型。

```
void EmployeeForm::addEmployee()
{
    int row = mapper->currentIndex();
    mapper->submit();
    tableModel->insertRow(row);
    mapper->setCurrentIndex(row);

    nameEdit->clear();
    extensionLineEdit->clear();
    startDateEdit->setDate(QDate::currentDate());
    nameEdit->setFocus();
}
```

当用户单击 Add 按钮时,就会调用 addEmployee()槽。我们首先重新找回当前行,因为它在提交后就遗失了。然后调用 submit()以确保对当前记录的修改没有任何遗失。尽管已经设置了自动提交策略 QDataWidgetMapper::AutoSubmit,仍必须手动提交。这是因为自动提交仅在用户改变光标焦点位置时才能使用——这样可以避免在每次用户插入或者删除一个字符时,频繁地对数据库进行更新操作——因为用户有可能刚编辑完一个字段,但当单击 Add 按钮时光标焦点位置并没有跳出该字段。接下来,我们插入一个新的空白行并让映射器导航浏览至该空白行。最后,初始化窗口部件,并将光标聚焦到用户开始键入的第一个就绪的窗口部件。

```
void EmployeeForm::deleteEmployee()
{
    int row = mapper->currentIndex();
    tableModel->removeRow(row);
    mapper->submit();
    mapper->setCurrentIndex(qMin(row, tableModel->rowCount() - 1));
}
```

对于删除操作,应从指明当前行开始,然后删除这一行并向数据库提交这个更改。必须手动提交删除的更改,因为自动提交策略仅适用于对记录做出的更改。最后,让映射器当前索引指向被删除行的下一行。如果删除的是最后一行,则指向最后一行。

QDataWidgetMapper 类使那些采用数据模型显示信息的数据可知型窗体的开发变得容易得多。在这个例子中,我们使用一个 QSqlRelationalTableModel 作为底层的数据模型,而 QDataWidgetMapper 则可以与任何数据模型一起使用,包括非 SQL 模型。另外一个可供选择的方式是直接使用 QSqlQuery 在窗体中填写数据,并更新数据库。这个方法要求更多的工作量,但也相应地更灵活一些。

下一节中,我们将看看 Staff Manager 应用程序的余下部分,包括在本节中开发的使用 EmployeeForm 类的代码。

13.4 在表中显示数据

在许多情况下,以表格式的视图为用户显示数据集是最简单的方法。本节给出 Staff Manager 应用程序的主窗体,它由两个呈主-从关系的 QTableView 组成(窗体如图 13.4 所示)。主视图是一个单位部门的列表,从视图则为当前部门中的雇员列表。两个视图都使用了 QSqlRelationalTableModels,因为它们呈现出的两个数据库表都有外键字段。相关的 CREATE TABLE 声明可以参见 245 页的描述。

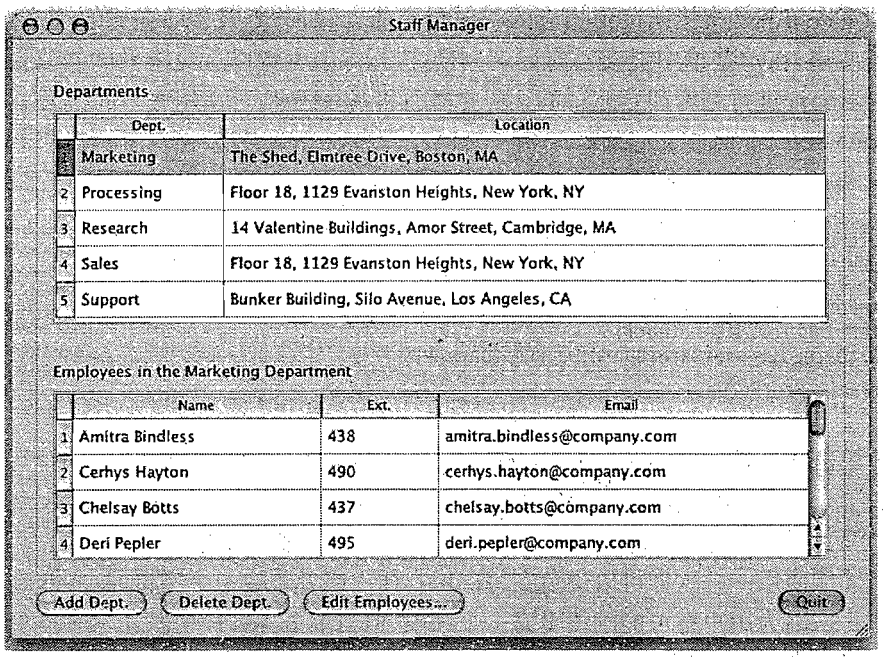


图 13.4 Staff Manger 应用程序

我们依然使用 enum 枚举类型变量为表的列索引号给出有具体含义的命名：

```
enum {
    Department_Id = 0,
    Department_Name = 1,
    Department_LocationId = 2
};
```

首先看看头文件中 MainForm 类的定义：

```
class MainForm : public QWidget
{
    Q_OBJECT

public:
    MainForm();

private slots:
    void updateEmployeeView();
    void addDepartment();
    void deleteDepartment();
    void editEmployees();

private:
    void createDepartmentPanel();
    void createEmployeePanel();

    QSqlRelationalTableModel *departmentModel;
    QSqlRelationalTableModel *employeeModel;
    QWidget *departmentPanel;
    ...
    QDialogButtonBox *buttonBox;
};
```

为了设立主-从关系，必须确保当用户浏览到主视图中不同的记录(行)时，更新从视图中的表并显示相关的记录。这可以通过私有 updateEmployeeView()槽实现。其他三个槽[addDepartment()，deleteDepartment()和 editEmployees()槽]的功能则可望文生义，两个私有函数[createDepartmentPanel()和 createEmployeePanel()]则是构造函数的帮助函数。

大多数构造函数的代码都与创建用户界面接口相关,并且会设立合适的信号-槽联系。而我们只关注那些与数据库编程相关的部分。

```
MainForm::MainForm()
{
    createDepartmentPanel();
    createEmployeePanel();
```

构造函数首先调用两个帮助函数 createDepartmentPanel()和 createEmployeePanel()。第一个函数创建并设立部门模型和视图,第二个函数则创建并设立雇员的模型和视图。在查看完构造函数以后,我们将看看这两个函数的有关部分。

构造函数的下一个部分设立了包含两个表视图的切分窗口,同时还设立窗体按钮。我们将省略所有这方面的代码。

```
    ...
    connect(addButton, SIGNAL(clicked()), this, SLOT(addDepartment()));
    connect(deleteButton, SIGNAL(clicked()),
            this, SLOT(deleteDepartment()));
    connect(editButton, SIGNAL(clicked()), this, SLOT(editEmployees()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    ...
    departmentView->setCurrentIndex(departmentModel->index(0, 0));
}
```

我们建立对话框中按钮和槽的连接,同时确保第一个部门就是当前的项。

既然看过了构造函数,接下来就看看 createDepartmentPanel()帮助函数中的代码,它设立了部门的模型和视图:

```
void MainForm::createDepartmentPanel()
{
    departmentPanel = new QWidget;

    departmentModel = new QSqlRelationalTableModel(this);
    departmentModel->setTable("department");
    departmentModel->setRelation(Department_LocationId,
            QSqlRelation("location", "id", "name"));
    departmentModel->setSort(Department_Name, Qt::AscendingOrder);
    departmentModel->setHeaderData(Department_Name, Qt::Horizontal,
                                   tr("Dept."));
    departmentModel->setHeaderData(Department_LocationId,
                                   Qt::Horizontal, tr("Location"));
    departmentModel->select();

    departmentView = new QTableView;
    departmentView->setModel(departmentModel);
    departmentView->setItemDelegate(new QSqlRelationalDelegate(this));
    departmentView->setSelectionMode(
            QAbstractItemView::SingleSelection);
    departmentView->setSelectionBehavior(QAbstractItemView::SelectRows);
    departmentView->setColumnHidden(Department_Id, true);
    departmentView->resizeColumnsToContents();
    departmentView->horizontalHeader()->setStretchLastSection(true);

    departmentLabel = new QLabel(tr("Depar&tments"));
    departmentLabel->setBuddy(departmentView);

    connect(departmentView->selectionModel(),
            SIGNAL(currentRowChanged(const QModelIndex &,
                                     const QModelIndex &)),
            this, SLOT(updateEmployeeView()));
    ...
}
```

这段代码与前一节中设立 employee 雇员表模型时所见到的代码相似。这个视图是一个标准

QTableView,但是因为外键的存在,必须使用 QSqlRelationalDelegate,这样外键的文本才能正常出现在视图中并能通过组合框进行更改,而不只是更改原始的 ID 号。

我们已经选择隐藏部门的 ID 字段,因为它对用户来说没有什么意义。我们也拉伸主视图中最后一列可视字段,也就是部门地址字段,以填满视图中水平方向上的可用空间。

部门主视图有它自己的针对 QAbstractItemView::SingleSelection 的模式设置选择以及针对 QAbstractItemView::SelectRows 的状态设置选择。模式设置意味着用户可以浏览到表中单独的单元格,而状态设置则表示当用户浏览记录项时,整行都是高亮显示的。

我们从视图的选择模型来建立 updateEmployeeView()槽与 currentRowChanged()信号的连接。这个连接是使主-从关系起作用的关键,它同时也确保了雇员从视图总是能显示部门主视图中被高亮选中的部门所对应的雇员信息。

createEmployeePanel()帮助函数内部的代码与 createDepartmentPanel()帮助函数的相似,但是也有一些重要的区别:

```
void MainForm::createEmployeePanel()
{
    employeePanel = new QWidget;
    employeeModel = new QSqlRelationalTableModel(this);
    employeeModel->setTable("employee");
    employeeModel->setRelation(Employee_DepartmentId,
            QSqlRelation("department", "id", "name"));
    employeeModel->setSort(Employee_Name, Qt::AscendingOrder);
    employeeModel->setHeaderData(Employee_Name, Qt::Horizontal,
                                 tr("Name"));
    employeeModel->setHeaderData(Employee_Extension, Qt::Horizontal,
                                 tr("Ext."));
    employeeModel->setHeaderData(Employee_Email, Qt::Horizontal,
                                 tr("Email"));

    employeeView = new QTableView;
    employeeView->setModel(employeeModel);
    employeeView->setSelectionMode(QAbstractItemView::SingleSelection);
    employeeView->setSelectionBehavior(QAbstractItemView::SelectRows);
    employeeView->setEditTriggers(QAbstractItemView::NoEditTriggers);
    employeeView->horizontalHeader()->setStretchLastSection(true);
    employeeView->setColumnHidden(Employee_Id, true);
    employeeView->setColumnHidden(Employee_DepartmentId, true);
    employeeView->setColumnHidden(Employee_StartDate, true);

    employeeLabel = new QLabel(tr("E&mployees"));
    employeeLabel->setBuddy(employeeView);
    ...
}
```

雇员视图的编辑触发器被设置为 QAbstractItemView::NoEditTriggers,它能有效确保视图的只读属性。在这个应用程序中,通过单击 Edit Employees 按钮,用户可以添加、编辑和删除雇员记录,而该按钮调用了前一节中开发的 EmployeeForm。

这次,我们隐藏三列,而不是仅隐藏一列。我们隐藏 id 列,因为它对于用户来说还是没有什么意义。还需要隐藏 departmentid 列,因为每一次显示的雇员信息都是属于当前选中部门的。最后,隐藏 startdate 列,因为它几乎没有任何实质作用,而且还可以通过单击 Edit Employees 按钮读取它。

```
void MainForm::updateEmployeeView()
{
    QModelIndex index = departmentView->currentIndex();
    if (index.isValid()) {
        QSqlRecord record = departmentModel->record(index.row());
        int id = record.value("id").toInt();
        employeeModel->setFilter(QString("departmentid = %1").arg(id));
```

```
        employeeLabel->setText(tr("E&mployees in the %1 Department")
                               .arg(record.value("name").toString()));
    } else {
        employeeModel->setFilter("departmentid = -1");
        employeeLabel->setText(tr("E&mployees"));
    }
    employeeModel->select();
    employeeView->horizontalHeader()->setVisible(
            employeeModel->rowCount() > 0);
}
```

只要当前部门发生更改(包括启动的时候),这个槽就会被调用。如果它是一个有效的当前部门,函数会重新找回该部门的 ID 号并且对其雇员模型建立一个过滤器程序。这就迫使雇员信息只对那些与其匹配的部门 ID 外键才显示。(过滤器程序就是没有 WHERE 关键字的 WHERE 子句)。我们还需要更新 employee 表上的标签文本以显示雇员所属部门的名称。

如果没有有效的部门名(比如数据库为空时),就将过滤器程序设置为与一个不存在的部门 ID 相匹配,这样就没有与其匹配的记录。

然后,对模型调用 select()函数以应用这个过滤器程序。这将依次发射信号,而视图将通过自我更新对该信号做出回应。最后,显示或者隐藏 employee 表的列标题,这取决于是否有任何雇员信息被显示。

```
void MainForm::addDepartment()
{
    int row = departmentModel->rowCount();
    departmentModel->insertRow(row);
    QModelIndex index = departmentModel->index(row, Department_Name);
    departmentView->setCurrentIndex(index);
    departmentView->edit(index);
}
```

如果用户单击"Add Dept."按钮,就会在 department 表的最后插入一个新行,并让这一行成为当前行,同时启动部门名那一列的编辑功能,就好像用户按下了 F2 键或者双击了该列一样。如果需要提供一些默认值,则将在调用 insertRow()后立即调用 setData()来完成。

我们不必考虑为新记录创建独特的键,因为使用的是一个自动增加列。如果这种方法不能或者不适合使用,则可以连接模型的 beforeInsert()信号。这个信号在用户编辑后将发射,正好在数据库的插入发生之前。这是放入 ID 或者处理用户数据的最佳时间。beforeDelete()和 beforeUpdate()信号很类似,它们对创建审计追踪非常有用。

```
void MainForm::deleteDepartment()
{
    QModelIndex index = departmentView->currentIndex();
    if (!index.isValid())
        return;

    QSqlDatabase::database().transaction();
    QSqlRecord record = departmentModel->record(index.row());
    int id = record.value(Department_Id).toInt();
    int numEmployees = 0;

    QSqlQuery query(QString("SELECT COUNT(*) FROM employee "
                            "WHERE departmentid = %1").arg(id));
    if (query.next())
        numEmployees = query.value(0).toInt();
    if (numEmployees > 0) {
        int r = QMessageBox::warning(this, tr("Delete Department"),
                    tr("Delete %1 and all its employees?")
                    .arg(record.value(Department_Name).toString()),
                    QMessageBox::Yes | QMessageBox::No);
        if (r == QMessageBox::No) {
```

```
            QSqlDatabase::database().rollback();
            return;
        }

        query.exec(QString("DELETE FROM employee "
                           "WHERE departmentid = %1").arg(id));
    }

    departmentModel->removeRow(index.row());
    departmentModel->submitAll();
    QSqlDatabase::database().commit();

    updateEmployeeView();
    departmentView->setFocus();
}
```

如果想删除部门且该部门没有任何雇员记录,这个操作就可以直接执行而没有其他任何确认提示信息。如果该部门中有雇员,就要求用户确认其删除操作。而如果用户确认其删除操作,就执行级联删除以确保数据库的关系完整性。为了实现这个功能,特别是针对那些并不会为我们保证关系完整性的数据库(如 SQLite 3),必须使用事务处理。

一旦启动事务,就执行一个查询以查明该部门中有多少雇员。只要有一名雇员,就弹出一个消息框要求其确认操作。如果用户单击 No,就回滚该事务并返回。否则,就删除该部门以及该部门中的所有雇员的信息,并且提交事务。

```
void MainForm::editEmployees()
{
    int employeeId = -1;
    QModelIndex index = employeeView->currentIndex();
    if (index.isValid()) {
        QSqlRecord record = employeeModel->record(index.row());
        employeeId = record.value(Employee_Id).toInt();
    }

    EmployeeForm form(employeeId, this);
    form.exec();
    updateEmployeeView();
}
```

只要用户单击 Edit Employees 按钮,就会调用 editEmployees()槽。首先分配一个无效的雇员 ID,然后用当前的雇员 ID 复写这个无效的雇员 ID。接着,构建 EmployeeForm 并让它在形式上显示出来。最后,调用 updateEmployeeView()槽,以使主窗体的从表视图自我刷新,因为此时雇员信息可能已经发生了更改。

本章介绍了 Qt 的模型/视图类如何让 SQL 数据库中查看和编辑数据的操作变得尽可能简单。当我们想使用一个窗体视图显示记录项时,可以利用 QDataWidgetMapper 将用户接口界面中的窗体部件映射到数据库记录的字段。主-从关系的设置也相当容易,仅仅要求一个信号-槽连接以及一个简单槽的实现。下钻型的窗体视图很直观,我们只需要导航浏览到主窗体构造函数中被选中的记录,就可以在从视图中看到与其对应的具体信息。如果没有记录被选中,则直接看到的就是第一条记录。

第14章 多 线 程

传统的图形用户界面应用程序都只有一个执行线程，并且一次只执行一个操作。如果用户从用户界面中调用一个比较耗时的操作，那么当执行这个操作时，虽然实际上该操作正在进行，但用户界面通常会冻结而不再响应。对于这个问题，第7章已经提供了一些解决方案。本章要讲述的多线程则是另外一种解决方案。

在多线程应用程序中，图形用户界面运行于它自己的线程中，而另外的事件处理过程则会发生在一个或多个其他线程中。这样做之后，即使在处理那些数据密集的事件时，应用程序也能对图形用户界面保持响应。当在一个单处理器上运行时，多线程应用程序可能会比实现同样功能的单线程应用程序运行得更慢一些，无法体现出其优势。但在目前多处理器系统越来越普及的情况下，多线程应用程序可以在不同的处理器中同时执行多个线程，从而获得更好的总体性能。

这一章将首先介绍如何子类化 QThread 以及如何使用 QMutex、QSemaphore 和 QWaitCondition，同步线程[①]。接着，我们将会看到当事件循环运行时，主线程和次线程之间是如何进行通信的。最后，会回顾一下哪些 Qt 类在次线程中可以使用，哪些类不可以使用。

多线程是一个很大的研究课题，有很多书籍专门介绍这一主题——例如 Bil Lewis 和 Daniel J. Berg 编著的“Threads Primer: A Guide to Multithreaded Programming”(Prentice Hall, 1995) 以及 Gregory Andrews 编著的“Multithreaded, Parallel, and Distributed Programming”(Addison-Wesley, 2000)。在这里，假设读者已经掌握了多线程编程的基础知识，因此本章的重点将放在如何开发多线程 Qt 应用程序方面，而并不过多地关注多线程这个主题本身。

14.1 创建线程

在 Qt 应用程序中提供多线程是非常简单的：只需要子类化 QThread 并且重新实现它的 run() 函数就可以了。为了显示这一过程是如何进行的，我们将从介绍一个非常简单的 QThread 子类的代码开始，它是一个可以在控制台上重复打印给定字符串的子类。应用程序的用户接口界面如图 14.1 所示。

```
class Thread : public QThread
{
    Q_OBJECT

public:
    Thread();

    void setMessage(const QString &message);
    void stop();

protected:
    void run();
```

① Qt 4.4 有望能提供一种更高级的线程应用编程接口，以完善这里讨论的线程类，从而使编写多线程应用程序时尽量少出错误。

```
private:
    QString messageStr;
    volatile bool stopped;
};
```

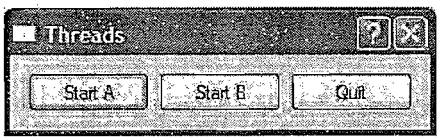


图 14.1 Threads 应用程序

这里的 Thread 类派生自 QThread 类,并且重新实现了 run()函数。这个类还额外提供了两个函数:setMessage()和 stop()。

stopped 被声明为易失性变量(volatile variable,断电或中断时数据丢失而不可再恢复的变量类型),这是因为不同的线程都需要访问它,并且我们也希望确保它能够在任何需要的时候都保持着最新读取的数值。如果省略 volatile 关键字,则编译器就会对这个变量的访问进行优化,那么就可能导致不正确的结果。

```
Thread::Thread()
{
    stopped = false;
}
```

在构造函数中把 stopped 设置为 false。

```
void Thread::run()
{
    while (!stopped)
        std::cerr << qPrintable(messageStr);
    stopped = false;
    std::cerr << std::endl;
}
```

在开始执行线程的时,就会调用 run()函数。只要 stopped 变量为 false 值,这个函数就会一直向控制台打印输出给定的消息。当控制离开 run()函数时,就会终止线程。

```
void Thread::stop()
{
    stopped = true;
}
```

stop()函数会把 stopped 变量设置为 true,从而告诉 run()停止向控制台输出文本。任何时候,stop()函数都可以由任一线程调用。考虑到这个实例的目的,我们假定对 bool 变量的赋值是一个原子操作。这是一个合理的假设,因为 bool 变量仅能有两种状态。在这一节的稍后部分,将会看到如何使用 QMutex 确保对一个变量的赋值成为一个原子操作。

QThread 提供了一个 terminate()函数,该函数可以在一个线程还在运行的时候就终止它的执行。我们不推荐使用 terminate(),这是因为它可以随时停止线程执行而不给这个线程自我清空的机会。一种更为安全的方法是使用 stopped 变量和 stop()函数,就像这里所做的那样。

下面将给出如何在一个简单的 Qt 应用程序中使用 Thread 类的示例,这个应用程序除了主线程之外还使用了两个线程,即线程 A 和线程 B。

```
class ThreadDialog : public QDialog
{
    Q_OBJECT

public:
```

```
    ThreadDialog(QWidget *parent = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void startOrStopThreadA();
    void startOrStopThreadB();

private:
    Thread threadA;
    Thread threadB;
    QPushButton *threadAButton;
    QPushButton *threadBButton;
    QPushButton *quitButton;
};
```

ThreadDialog 类声明了两个类型为 Thread 的变量和一些按钮，提供基本的用户界面。

```
ThreadDialog::ThreadDialog(QWidget *parent)
    : QDialog(parent)
{
    threadA.setMessage("A");
    threadB.setMessage("B");

    threadAButton = new QPushButton(tr("Start A"));
    threadBButton = new QPushButton(tr("Start B"));
    quitButton = new QPushButton(tr("Quit"));
    quitButton->setDefault(true);

    connect(threadAButton, SIGNAL(clicked()),
            this, SLOT(startOrStopThreadA()));
    connect(threadBButton, SIGNAL(clicked()),
            this, SLOT(startOrStopThreadB()));
    ...
}
```

在构造函数中，调用 setMessage()让第一个线程重复打印字母“A”，而让第二个线程重复打印字母“B”。

```
void ThreadDialog::startOrStopThreadA()
{
    if (threadA.isRunning()) {
        threadA.stop();
        threadAButton->setText(tr("Start A"));
    } else {
        threadA.start();
        threadAButton->setText(tr("Stop A"));
    }
}
```

当用户单击用于线程 A 的按钮时，如果这个线程正在运行，startOrStopThreadA()就让它停止运行；否则，就让它开始运行。startOrStopThreadA()同时还更新该按钮上的文本。

```
void ThreadDialog::startOrStopThreadB()
{
    if (threadB.isRunning()) {
        threadB.stop();
        threadBButton->setText(tr("Start B"));
    } else {
        threadB.start();
        threadBButton->setText(tr("Stop B"));
    }
}
```

startOrStopThreadB()的代码与 startOrStopThreadA()的代码在结构上基本一致。

```
void ThreadDialog::closeEvent(QCloseEvent *event)
{
    threadA.stop();
    threadB.stop();
    threadA.wait();
    threadB.wait();
    event->accept();
}
```

如果用户单击了 Quit 或者关闭了窗口,就停止所有正在运行的线程,并且在调用函数 QCloseEvent::accept()之前等待它们完全结束[可使用 QThread::wait()]。这样就可以确保应用程序是以一种原始清空的状态而退出的,尽管在这个实例中是否这样做并不要紧。

如果运行这个应用程序,并且单击按钮"Start A",那么控制台终端就会被连续输出的字母"A"填满。如果再单击按钮"Start B",那么控制台终端就会被以交替顺序输出的字母"A"和"B"填满。如果再单击按钮"Stop A",那么就将只输出字母"B"。

14.2 同步线程

对于多线程应用程序,一个最基本要求就是能实现几个线程的同步执行。Qt 提供了以下这几个用于同步的类:QMutex、QReadWriteLock、QSemaphore 和 QWaitCondition。

QMutex 类提供了一种保护一个变量或者一段代码的方法,这样就可以每次只让一个线程读取它。这个类提供了一个 lock()函数来锁住互斥量(mutex)。如果互斥量是解锁的(unlock),那么当前线程就立即占用并锁定(lock)它;否则,当前线程就会被阻塞,直到掌握这个互斥量的线程对它解锁为止。以上述两种方式中的任意一种对 lock()调用返回时,当前线程都会保持这个互斥量,直到调用 unlock()为止。QMutex 类还提供了一个 tryLock()函数,如果该互斥量已经锁住,它就会立即返回。

例如,假设我们想使用 QMutex 来保护前一节中 Thread 类的 stopped 变量,则应当向 Thread 类中添加如下的成员变量:

```
private:
    ...
    QMutex mutex;
};
```

函数 run()也需要相应修改:

```
void Thread::run()
{
    forever {
        mutex.lock();
        if (stopped) {
            stopped = false;
            mutex.unlock();
            break;
        }
        mutex.unlock();

        std::cerr << qPrintable(messageStr);
    }
    std::cerr << std::endl;
}
```

stop()函数则需要修改为:

```
void Thread::stop()
{
    mutex.lock();
    stopped = true;
    mutex.unlock();
}
```

在一些复杂函数或是在抛出 C++ 异常的函数中锁定和解锁互斥量，非常容易发生错误。Qt 提供了方便的 QMutexLocker 类来简化对互斥量的处理。QMutexLocker 的构造函数接受 QMutex 作为参数并且将其锁住。QMutexLocker 的析构函数则对这个互斥量进行解锁。例如，我们可以重新编写之前介绍过的 run()和 stop()函数如下：

```
void Thread::run()
{
    forever {
        {
            QMutexLocker locker(&mutex);
            if (stopped) {
                stopped = false;
                break;
            }
        }

        std::cerr << qPrintable(messageStr);
    }
    std::cerr << std::endl;
}

void Thread::stop()
{
    QMutexLocker locker(&mutex);
    stopped = true;
}
```

使用互斥量的一个问题在于：每次只能有一个线程可以访问同一变量。在程序中，可能会有许多线程同时尝试访问读取某一变量(不是修改该变量)，此时，互斥量可能就会成为一个严重的性能瓶颈。在这种情况下，可以使用 QReadWriteLock，它是一个同步类，允许同时执行多个读取访问而不会影响性能。

在 Thread 类中，用 QReadWriteLock 替换 QMutex 来保护 stopped 变量毫无意义，因为在任意给定时刻，最多只有一个线程会试图读取这个变量。一个更为恰当的实例，是在程序中应当包含一个或多个访问某些共享数据的阅读程序线程，并且还有一个或多个修改这些共享数据的写入程序线程。例如：

```
MyData data;
QReadWriteLock lock;

void ReaderThread::run()
{
    ...
    lock.lockForRead();
    access_data_without_modifying_it(&data);
    lock.unlock();
    ...
}

void WriterThread::run()
{
    ...
    lock.lockForWrite();
    modify_data(&data);
    lock.unlock();
    ...
}
```

为简便起见，我们可以使用 QReadLocker 类和 QWriteLocker 类对 QReadWriteLock 进行锁定和解锁。

QSemaphore 是互斥量的另外一种泛化表示形式，但与读-写锁定不同，信号量(semaphore)可以用于保护(guard)一定数量的相同资源。下面的两小段程序代码给出了 QSemaphore 和 QMutex 之间

的对应关系：

```
QSemaphore semaphore(1);
semaphore.acquire();
semaphore.release();
```

```
QMutex mutex;
mutex.lock();
mutex.unlock();
```

通过把 1 传递给构造函数，就告诉这个信号量它控制了一个单一的资源。使用信号量的优点是可以传递除 1 之外的数字给构造函数，然后可以多次调用 acquire()来获取大量资源。

一个典型的信号量应用程序是当两个线程间传递一定量的数据(DataSize)时，这两个线程会使用某一特定大小(BufferSize)的共享环形缓冲器(circular buffer)：

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];
```

生产者线程向缓冲器中写入数据，直到它到达缓冲器的终点为止；然后它会再次从起点重新开始，覆盖已经存在的数据。消费者线程则会读取生成的数据。图 14.2 阐明了这一情况，其中假设使用的只是一个很小的 16 字节缓冲器。

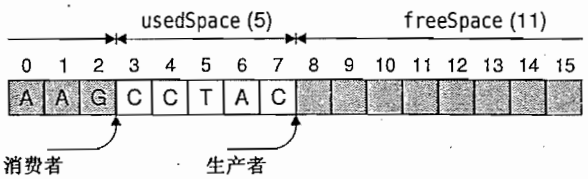


图 14.2 生产者-消费者模型

在生产者-消费者实例中对于同步的需求有两个部分：如果生产者线程生成数据的速度太快，那么将会把消费者线程还没有读取的数据覆盖掉；如果消费者线程读取数据的速度过快，那么它将会跃过生产者线程而读取一些垃圾数据。

解决这一问题的一个粗略的方法是让生产者线程填满缓冲器，然后等待消费者线程读取完缓冲器中全部数据为止。然而，在多处理器的机器上，让生产者和消费者两个线程分别同时操作缓冲器的两个部分则要比前面的方案快得多。

因此，解决这一问题的一个更为有效的方法是使用两个信号量：

```
QSemaphore freeSpace(BufferSize);
QSemaphore usedSpace(0);
```

freeSpace 信号量控制生产者线程写入数据的那部分缓冲器，usedSpace 信号量则控制消费者线程读取数据的那部分缓冲器区域。这两个区域是相互补充的。我们用 BufferSize(4096)初始化 freeSpace 信号量，这也就意味着它至多可以获得 4096 字节的缓冲器资源。在启动这个应用程序时，阅读程序线程(reader thread)将会开始获得“自由的”(free)字节并且把它们转换为“用过的”(used)字节。我们用 0 初始化 usedSpace 信号量，以确保消费者线程不会在一开始就读取到垃圾数据。

对于这个实例来说，每一个字节都计为一个资源。在实际使用的应用程序中，很可能会在更大的单元(例如，一次 64 字节或 256 字节)上进行操作，这样可以相应减少信号量的开销。

```
void Producer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        freeSpace.acquire();
        buffer[i % BufferSize] = "ACGT"[uint(std::rand()) % 4];
        usedSpace.release();
    }
}
```

在生产者线程中，每次反复写入都是从获取一个“自由”字节开始的。如果该缓冲器中充满了消费者线程还没有读取的数据，那么对 acquire()的调用就会被阻塞，直到消费者线程开始“消费”这些数据为止。一旦获取了这一字节，就用一些随机数据(如“A”、“C”、“G”或者“T”)进行写入，并且把这个字节释放为“用过的”字节，以便让消费者线程读取到。

```
void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        usedSpace.acquire();
        std::cerr << buffer[i % BufferSize];
        freeSpace.release();
    }
    std::cerr << std::endl;
}
```

在消费者线程中，我们从获取一个“用过的”字节开始。如果缓冲器中没有任何可读的数据，那么将会阻塞对 acquire()调用，直到生产者线程生成一些可读的数据为止。一旦获取到这个字节，就打印出它并且把这个字节释放为“自由的”字节，这样生产者线程就可以再次使用其他的数据复写它。

```
int main()
{
    Producer producer;
    Consumer consumer;
    producer.start();
    consumer.start();
    producer.wait();
    consumer.wait();
    return 0;
}
```

最后，在 main()中，我们开启生产者和消费者线程。然后，生产者线程把一些“自由的”空间转换成“用过的”空间，而消费者线程则可以把它再次转换为“自由的”空间。

当运行这个程序时，它会把一个含有 100 000 个“A”、“C”、“G”和“T”的随机序列写入控制台中，然后终止。为了真正理解到底发生了什么，我们可以禁用写入输出数据，而让生产者线程在每次产生一个字节时输出一个“P”，让消费者线程在每次读取一个字节时输出一个“c”。为了更加简单地跟踪这一过程，我们可以使用较小的 DataSize 和 BufferSize 值。

例如，这里给出的是当使用 DataSize 等于 10 且 BufferSize 等于 4 时的可能的运行结果：“PcPcPcPcPcPcPcPcPcPc”。在这种情况下，只要生产者线程一产生字节，消费线程者就会迅速读取它。也就是说，这两个线程的执行速度是一样的。另外一种可能是，在消费者开始读取缓存之前，生产者线程就已经写满了整个缓冲器，这样输出的结果就会是：“PPPPccccPPPPccccPPcc”。还有很多其他可能的情形。信号为系统特定的线程调度程序提供了很大程度上的自由，利用线程调度程序可以研究这些线程的行为并且可以用于选择一个适当的调度策略。

对于使生产者和消费者线程同步的这个问题，另一解决方案是使用 QWaitCondition 和 QMutex。QWaitCondition 允许一个线程在满足一定的条件下触发其他多个线程。这样就可以比只使用互斥量提供更为精确的控制。为了说明它是如何工作的，我们将会使用等待条件来重新完成上述的生产者-消费者例子。

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];

QWaitCondition bufferIsNotFull;
QWaitCondition bufferIsNotEmpty;
QMutex mutex;
int usedSpace = 0;
```

除了缓冲器之外,我们还声明了两个 QWaitCondition、一个 QMutex 和一个变量,该变量用来存储在缓冲器中有多少个"用过的"字节。

```
void Producer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        while (usedSpace == BufferSize)
            bufferIsNotFull.wait(&mutex);
        buffer[i % BufferSize] = "ACGT"[uint(std::rand()) % 4];
        ++usedSpace;
        bufferIsNotEmpty.wakeAll();
        mutex.unlock();
    }
}
```

在生产者线程中,首先检查缓冲器是否已经写满。如果它已经被写满,就等待"缓冲器非写满"这一条件。当满足这个条件时,就向缓冲器中写入一个字节,增加"用过的空间"(usedSpace),并且触发任何一个等待"缓冲器非空"条件变为 true 值时的线程。

我们使用一个互斥量来保护所有对 usedSpace 变量的访问。QWaitCondition::wait()函数可以把锁定的互斥量作为它的第一个参数,在阻塞当前线程之前它会解锁,然后在返回之前它会锁定。

对于这个实例,可以把下面的 while 循环:

```
while (usedSpace == BufferSize)
    bufferIsNotFull.wait(&mutex);
```

替换为如下的 if 语句:

```
if (usedSpace == BufferSize) {
    mutex.unlock();
    bufferIsNotFull.wait();
    mutex.lock();
}
```

然而,一旦我们被允许使用多个生产者线程时,将会中断 if 语句,因为另外一个生产者可在 wait()调用之后立即占用这个互斥量并且可使这个"缓冲器非写满"条件再次变为 false 值。

```
void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        while (usedSpace == 0)
            bufferIsNotEmpty.wait(&mutex);
        std::cerr << buffer[i % BufferSize];
        --usedSpace;
        bufferIsNotFull.wakeAll();
        mutex.unlock();
    }
    std::cerr << std::endl;
}
```

消费者线程所做的和生产者正好相反:它等待"缓冲器非空"的条件并且触发正在等待"缓冲器非写满"条件的任意线程。

在目前为止提及的所有实例中,我们的线程都已经访问了这些相同的全局变量。但是在一些多线程的应用程序中,需要有一个在不同线程中保存不同数值的全局变量。这种变量通常称为线程本地存储(thread-local storage)或者线程特定数据(thread-specific data)。我们可以使用一个联通线程 ID 号[由 QThread::currentThread()返回]的映射来模仿它,但是更好的方法是使用 QThreadStorge <T>类。

QThreadStorge <T> 的一种常见用法是用于高速缓存中。通过在不同线程中拥有一个独立的高速缓存，就可以避免用于锁住、解锁和可能等待一个互斥量的计算开销。例如：

```
QThreadStorage<QHash<int, double> *> cache;

void insertIntoCache(int id, double value)
{
    if (!cache.hasLocalData())
        cache.setLocalData(new QHash<int, double>);
    cache.localData()->insert(id, value);
}

void removeFromCache(int id)
{
    if (cache.hasLocalData())
        cache.localData()->remove(id);
}
```

这里的 cache 变量在每一个线程中都保存一个指向 QMap <int, double> 的指针。（因为某些编译器的问题，QThreadStorage <T> 中的模板类型必须是指针类型。）在特定线程中第一次使用高速缓存时，hasLocalData()就会返回 false 值，而且我们可以创建一个 QHash <int, double> 对象。

除了高速缓存之外，QThreadStorage <T> 还可以用于全局错误状态变量（与 errno 相似），这样可以确保对某一线程的修改不会影响到其他的线程。

14.3　与主线程通信

当 Qt 应用程序开始执行时，只有主线程是在运行的。主线程是唯一允许创建 QApplication 或者 QCoreApplication 对象，并且可以对创建的对象调用 exec()的线程。在调用 exec()之后，这个线程或者等待一个事件，或者处理一个事件。

通过创建一些 QThread 子类的对象，主线程可以开始一些新的线程，就像上一节中所做的那样。如果这些新的线程需要在它们之间进行通信，则可以使用含有互斥量、读-写锁、信号或者等待条件的共享变量。但在这些技术中，没有任何一个可以用来与主线程进行通信，因为它们会锁住事件循环并且会冻结用户界面。

在次线程和主线程之间进行通信的一个解决方案是在线程之间使用信号-槽的连接。通常情况下，信号和槽机制可以同步操作，这就意味着在发射信号的时候，使用直接函数即可立刻调用连接到一个信号上的多个槽。

然而，当连接位于不同线程中的对象时，这一机制就会变得不同步起来[这种状态可以通过修改 QObject::connect()中的第 5 个可选参数而改变]。在底层，实际是通过置入一个事件来实现这些连接的。这个槽接着就会由线程的事件循环调用，而在该线程中存在着接收器对象。在默认情况下，QObject 存在于创建它的线程中，通过调用 QObject::moveToThread()可以在任意时刻修改它。

为了描绘线程之间的信号-槽连接是如何工作的，我们将查看一下在 Image Pro 应用程序中的代码。该程序是一个基本的图像处理应用程序，允许用户旋转图片、重定义图片的大小以及改变图片的色彩深度。这个应用程序（如图 14.3 所示）使用一个次线程在不锁住事件循环的情况下执行对图片的操作。在处理非常大的图片时，这样做会在效果上产生很大的差别。次线程具有一系列的任务，或者是“事务”，可以用来完成事件并发送事件给主窗口以报告进度。

```
ImageWindow::ImageWindow()
{
    imageLabel = new QLabel;
    imageLabel->setBackgroundRole(QPalette::Dark);
```

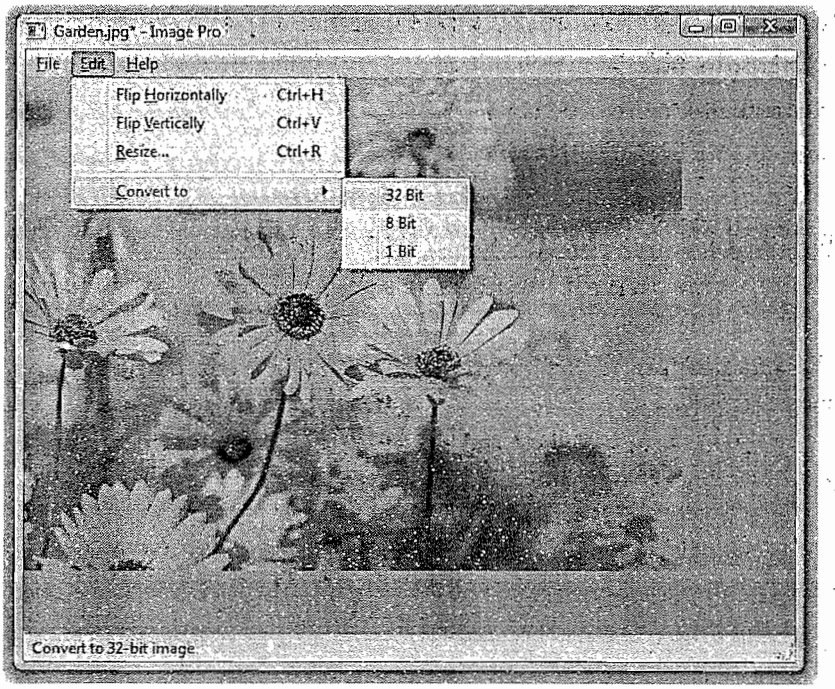


图 14.3 Image Pro 应用程序

```
    imageLabel->setAutoFillBackground(true);
    imageLabel->setAlignment(Qt::AlignLeft | Qt::AlignTop);
    setCentralWidget(imageLabel);

    createActions();
    createMenus();

    statusBar()->showMessage(tr("Ready"), 2000);

    connect(&thread, SIGNAL(transactionStarted(const QString &)),
            statusBar(), SLOT(showMessage(const QString &)));
    connect(&thread, SIGNAL(allTransactionsDone()),
            this, SLOT(allTransactionsDone()));

    setCurrentFile("");
}
```

在 ImageWindow 构造函数中值得注意的部分是这两个信号-槽连接。它们两个都与 transactionThread 对象发射的信号直接有关。稍后将介绍 TransactionThread 对象。

```
void ImageWindow::flipHorizontally()
{
    addTransaction(new FlipTransaction(Qt::Horizontal));
}
```

flipHorizontally()槽用来创建一个“翻转”事务并且用私有函数 addTransaction()注册它。而 flipVertically()、resizeImage()、convertTo32Bit()、convertTo8Bit()和 convertTo1Bit()这几个函数都很相似。

```
void ImageWindow::addTransaction(Transaction *transact)
{
    thread.addTransaction(transact);
    openAction->setEnabled(false);
    saveAction->setEnabled(false);
    saveAsAction->setEnabled(false);
}
```

addTransaction()函数会向次线程的事务队列中添加一个事务,并且当正在处理这些事务时禁用 Open、Save 和 Save As 等操作。

```
void ImageWindow::allTransactionsDone()
{
    openAction->setEnabled(true);
    saveAction->setEnabled(true);
    saveAsAction->setEnabled(true);
    imageLabel->setPixmap(QPixmap::fromImage(thread.image()));
    setWindowModified(true);
    statusBar()->showMessage(tr("Ready"), 2000);
}
```

当 transactionThread 的事务队列变为空时,就会调用这个 allTransactionsDone()槽。

现在,我们来看看 TransactionThread 类。与大多数 QThread 子类相似,TransactionThread 类的实现是需要慎重对待的,因为 run()函数在它自己的线程内执行,其他函数(包含构造函数和析构函数)则都从主线程中调用。该类的定义如下:

```
class TransactionThread : public QThread
{
    Q_OBJECT

public:
    TransactionThread();
    ~TransactionThread();

    void addTransaction(Transaction *transact);
    void setImage(const QImage &image);
    QImage image();

signals:
    void transactionStarted(const QString &message);
    void allTransactionsDone();

protected:
    void run();

private:
    QImage currentImage;
    QQueue<Transaction *> transactions;
    QWaitCondition transactionAdded;
    QMutex mutex;
};
```

TransactionThread 类维护着一个事务队列,并且在后台一个接一个地处理及执行它们。在上述代码的私有[private()]部分段,声明了 4 个成员变量:

- currentImage 保存事务应用的图片。
- transactions 是待处理事务的队列。
- transactionAdded 是一个等待条件,当有新的事务添加到队列中时,用于触发线程。
- mutex 用于防止 currentImage 和 transactions 成员变量并发访问。

这里给出类的构造函数如下:

```
TransactionThread::TransactionThread()
{
    start();
}
```

在构造函数中,只调用 QThread::start()以启动将执行事务的线程。

```
TransactionThread::~TransactionThread()
{
    {
        QMutexLocker locker(&mutex);
        while (!transactions.isEmpty())
            delete transactions.dequeue();
        transactions.enqueue(EndTransaction);
        transactionAdded.wakeOne();
    }

    wait();
}
```

在析构函数中,我们清空事务队列,并且给队列添加一个专门的 EndTransaction 标记。然后,在基类析构函数被隐式调用之前,触发线程并使用 QThread::wait()等待到它结束为止。如果没有调用 wait()函数,则当线程尝试读取类的成员变量时极有可能导致失败。

在调用 wait()函数之前,QMutexLocker 的析构函数会在其内部程序模块的末尾解锁互斥量。调用 wait()函数之前解锁互斥量是非常重要的;否则,程序将以死锁的情况结束:次线程将永久地等待互斥量被解锁,而主线程则在进一步执行之前保持互斥量并等待次线程执行结束。

```
void TransactionThread::addTransaction(Transaction *transact)
{
    QMutexLocker locker(&mutex);
    transactions.enqueue(transact);
    transactionAdded.wakeOne();
}
```

addTransaction()函数会把事务添加到事务队列中,并且如果事务线程还没有运行,就触发它。所有对 transactions 成员变量的访问都由一个互斥量保护起来,因为主线程或许会在次线程遍历所有事务的同时通过 addTransaction()修改这些成员变量。

```
void TransactionThread::setImage(const QImage &image)
{
    QMutexLocker locker(&mutex);
    currentImage = image;
}

QImage TransactionThread::image()
{
    QMutexLocker locker(&mutex);
    return currentImage;
}
```

setImage()和 image()函数可以让主线程设立一个图片,在该图片上可以执行事务处理,并且可以在所有事务处理完毕时找回最终的结果图片。

```
void TransactionThread::run()
{
    Transaction *transact = 0;
    QImage oldImage;

    forever {
        {
            QMutexLocker locker(&mutex);

            if (transactions.isEmpty())
                transactionAdded.wait(&mutex);
            transact = transactions.dequeue();
            if (transact == EndTransaction)
                break;

            oldImage = currentImage;
```

```
        }

        emit transactionStarted(transact->message());
        QImage newImage = transact->apply(oldImage);
        delete transact;

        {
            QMutexLocker locker(&mutex);
            currentImage = newImage;
            if (transactions.isEmpty())
                emit allTransactionsDone();
        }
    }
}
```

函数 run()会遍历事务队列,并且可以通过对它们调用 apply()函数依次执行每一个事务,直到读到 EndTransaction 标记为止。如果事务队列为空,则线程在"事务添加"条件下等待。

在开始执行事务之前,我们会发射 transactionStarted()信号并在这个应用程序的状态栏上显示相应的消息。当所有的事务都已经处理完毕,会发射 allTransactionsDone()信号。

```
class Transaction
{
public:
    virtual ~Transaction() { }

    virtual QImage apply(const QImage &image) = 0;

    virtual QString message() = 0;
};
```

Transaction 类是一个用户可以对图片进行相关操作的抽象基类。需要通过一个 Transaction 指针来删除 Transaction 子类的实例,因此虚析构函数是必要的。Transaction 类有三个具体子类:FlipTransaction、ResizeTransaction 和 ConvertDepthTransaction。这里将只查看 FlipTransaction 子类,其他两个子类基本与此相似。

```
class FlipTransaction : public Transaction
{
public:
    FlipTransaction(Qt::Orientation orientation);

    QImage apply(const QImage &image);
    QString message();

private:
    Qt::Orientation orientation;
};
```

FlipTransaction 的构造函数带有一个参数,该参数用来指定翻转方向(水平翻转或垂直翻转)。

```
QImage FlipTransaction::apply(const QImage &image)
{
    return image.mirrored(orientation == Qt::Horizontal,
                          orientation == Qt::Vertical);
}
```

apply()函数对 QImage 调用 QImage::mirrored(),它既作为接收参数也是返回值。

```
QString FlipTransaction::message()
{
    if (orientation == Qt::Horizontal) {
        return QObject::tr("Flipping image horizontally...");
    } else {
        return QObject::tr("Flipping image vertically...");
    }
}
```

message()函数返回正在处理的操作的消息,这些消息会显示在状态栏中。当发射 transactionStarted()信号时,就会在 transactionThread::run()中调用这个函数。

Image Pro 应用程序展示了 Qt 的信号-槽机制是如何使主线程与次线程之间的通信变得简单易行的。实现次线程时要谨慎对待,因为必须使用互斥量来保护成员变量,而且必须使用一个等待条件以在适当的时候停止或者触发线程。*Qt Quarterly* 中"Monitors and Wait Conditions in Qt"系列文章的 http://doc.trolltech.com/qq/qq21-monitors.html 和 http://doc.trolltech.com/qq/qq22-monitors2.html,给出了一些关于如何开发和测试那些使用互斥量和等待条件来保证同步的 QThread 子类的思想。

14.4 在次线程中使用 Qt 的类

当函数可以同时被不同的线程安全地调用时,就称其为"线程安全的"(thread-safe)。如果在不同的线程中对某一共享数据同时调用两个线程安全的函数,那么结果将总是可以确定的。若将这个概念推广,当一个类的所有函数都可以同时被不同的线程调用,并且它们之间互不干涉,即使是在操作同一个对象的时候也互不妨碍,我们就把这个类称为是线程安全的。

Qt 中,线程安全的类有 QMutex、QMutexLocker、QReadWriteLock、QReadLocker、QWriteLocker、QSemaphore、QThreadStorage <T> 以及 QWaitCondition。此外,部分 QThread 应用编程接口和其他某些函数也是线程安全的,特别是 QObject::connect()、QObject::disconnect()、QCoreApplication::postEvent()以及 QCoreApplication::removePostedEvents()。

绝大多数 Qt 的非图形用户界面类都符合一个并不太严格的要求:它们都必须是可重入的(reentrant)。如果类的不同实例可同时用于不同的线程,那么这个类就是可重入的。然而,在多个线程中同时访问同一个可重入的对象是不安全的,而是应该使用一个互斥量来保护这个类的访问。一个类是否是可重入的,在 Qt 的参考文档中有标记。通常情况下,任何没有被全局引用或者被其他共享数据引用的 C++ 类都认为是可重入的。

QObject 是可重入的,但有必要记住它的三个约束条件:

1. QObject 的子对象必须在它的父对象线程中创建

特别需要说明的是,这一约束条件意味着在次线程中创建的对象永远不能将 QThread 对象作为创建它们的父对象,因为 QThread 对象是在另外一个线程(主线程或者另外一个不同的次线程)中创建的。

2. 在删除对应的 QThread 对象之前,必须删除所有在次线程中创建的 QObject 对象

通过在 QThread::run()中的堆栈上创建这些对象,就可以完成这一点。

3. 必须在创建 QObject 对象的线程中删除它们

如果需要删除一个存在于不同线程中的 QObject 对象,则必须调用线程安全的 QObject::deleteLater()函数,它可以置入一个"延期删除"(deferred delete)事件。

一些非图形用户界面的 QObject 子类,如 QTimer、QProcess,以及用于网络方面的类都是可重入的。只要任意线程处于事件循环中,就可以在这个线程中使用它们。对于次线程,通过调用 QThread::exec()或者其他方便函数,如 QProcess::waitForFinished()和 QAbstractSocket::waitForDisconnected(),都可以启动一个这样的事件循环。

由于从那些为 Qt 的图形用户界面支持提供编译的低级库上继承的局限性,QWidget 和它的子类都是不可重入的。这样造成的后果之一就是我们不能在一个来自次线程的窗口部件上直接调用函数。打个比方说,如果想从次线程中修改一个 QLabel 中的文本,则可以发射一个连接到 QLa-

bel::setText()的信号,或者从该线程中调用 QMetaObject::invokeMethod()。例如:

```
void MyThread::run()
{
    ...
    QMetaObject::invokeMethod(label, SLOT(setText(const QString &)),
                              Q_ARG(QString, "Hello"));
    ...
}
```

很多 Qt 的非图形用户界面类,包括 QImage、QString 和一些容器类,都使用了隐式共享作为一项优化技术。虽然这样的优化通常会让类变成不可重入的,但在 Qt 中这不是一个大问题,因为 Qt 使用原子汇编语言指令(atomic assembly language instruction)来实现线程安全引用计数,这可以让 Qt 的隐式共享类变成可重入的。

Qt 的 QtSql 模块也可用于多线程的应用程序,但是它也有其约束条件,这个条件因数据库的不同而不同。有关的详细情况,请参考 http://doc.trolltech.com/4.3/sql-driver.html。有关多线程注意事项的一个完全列表,可以参考 http://doc.trolltech.com/4.3/threads.html。

第15章 网 络

Qt提供了QFtp和QHttp两个类与FTP和HTTP协议配合使用。这些协议的使用让文件的下载和上传变得更加容易。另外,HTTP协议也使向网站服务器发送请求并重新找回结果的过程变得简单易行。

Qt还提供了较低级的QTcpSocket和QUdpSocket类,它们将实现TCP和UDP传输协议。TCP是一个可靠的面向连接的协议,它按照网络节点间传输的数据流形式进行操作;UDP是一个不可靠的无连接协议,它主要基于网络节点间离散信息包的传输。这两个协议都可以用于创建网络客户端和服务器应用程序。若要创建服务器应用程序,还需要QTcpServer类来处理引入的TCP连接。我们可以使用QSslSocket代替QTcpSocket来建立安全的SSL/TLS连接。

15.1 写FTP客户端

QFtp类在Qt中实现了FTP协议的客户端程序,它提供了非常多的函数来执行多数常见的FTP操作,同时还可以执行任意的FTP指令。

QFtp类是异步工作的。若调用一个像get()或者put()这样的函数,它会立即返回并且仅在控制权回到Qt的事件循环时才发生数据传输。这样就确保了在执行FTP指令时,用户界面可以保持响应。

我们将从如何使用get()函数获取一个单一文件的例子开始讲起。该例子是一个名为ftpget的控制台应用程序,它可以下载指定命令行上的远程文件。首先看看main()主函数:

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QStringList args = QCoreApplication::arguments();

    if (args.count() != 2) {
        std::cerr << "Usage: ftpget url" << std::endl
                  << "Example:" << std::endl
                  << "    ftpget ftp://ftp.trolltech.com/mirrors"
                  << std::endl;
        return 1;
    }

    FtpGet getter;
    if (!getter.getFile(QUrl(args[1])))
        return 1;

    QObject::connect(&getter, SIGNAL(done()), &app, SLOT(quit()));

    return app.exec();
}
```

我们要创建一个QCoreApplication而不是它的子类QApplication,以避免连接到QtGui库。QCoreApplication::arguments()函数返回命令行参数作为一个QStringList列表,采用第一项作为程序被调用时的名称,同时删除掉诸如-style型的所有Qt指定的参数。main()函数的核心是构建FtpGet对象和getFile()调用。如果该函数调用成功,就会让事件循环一直运行下去,直到下载完毕。

FtpGet子类完成了所有的工作,其子类的定义如下:

```
class FtpGet : public QObject
{
    Q_OBJECT

public:
    FtpGet(QObject *parent = 0);

    bool getFile(const QUrl &url);

signals:
    void done();

private slots:
    void ftpDone(bool error);

private:
    QFtp ftp;
    QFile file;
};
```

这个类有一个公有函数 getFile(),可用来获取由 URL 指定的文件。QUrl 类提供了一个高级接口,用来提取 URL 的不同部分,如文件名称、路径、协议和端口。

FtpGet 有一个私有槽 ftpDone(),当文件传输完成时,就会调用它;而当文件下载完成后,它将发送一个 done()信号。这个类还有两个私有变量:ftp 变量和 file 变量,前者的类型为 QFtp,封装一个到 FTP 服务器的连接;后者用来向磁盘写入所下载的文件。

```
FtpGet::FtpGet(QObject *parent)
    : QObject(parent)
{
    connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
}
```

在构造函数中,我们把 QFtp::done(bool)信号与 ftpDone(bool)私有槽连接起来。当所有的请求都已处理完时,QFtp 就发射 done(bool)信号,bool 参数指明是否有错误发生。

```
bool FtpGet::getFile(const QUrl &url)
{
    if (!url.isValid()) {
        std::cerr << "Error: Invalid URL" << std::endl;
        return false;
    }

    if (url.scheme() != "ftp") {
        std::cerr << "Error: URL must start with 'ftp:'" << std::endl;
        return false;
    }

    if (url.path().isEmpty()) {
        std::cerr << "Error: URL has no path" << std::endl;
        return false;
    }

    QString localFileName = QFileInfo(url.path()).fileName();
    if (localFileName.isEmpty())
        localFileName = "ftpget.out";

    file.setFileName(localFileName);
    if (!file.open(QIODevice::WriteOnly)) {
        std::cerr << "Error: Cannot write file "
                  << qPrintable(file.fileName()) << ": "
                  << qPrintable(file.errorString()) << std::endl;
        return false;
    }

    ftp.connectToHost(url.host(), url.port(21));
    ftp.login();
    ftp.get(url.path(), &file);
```

```
    ftp.close();
    return true;
}
```

getFile()函数首先检查传入的 URL。如果遇到问题,该函数将打印出一条出错信息到 cerr,同时返回 false 以表明下载失败。

通过对 ftpget.out 的备份,我们可以试图利用 URL 自身创建一个合理的本地文件名而不是让用户自己虚构一个文件名。如果打开文件失败,就会打印出错信息并返回 false。

接着,利用 QFtp 对象执行一个含 4 个 FTP 指令的序列。url.port(21)调用返回在 URL 中指定的端口号;如果在 URL 中并未指定任何端口号,则返回端口 21。因为没有给 login()函数任何用户名或者密码,所以此处需要一个匿名登录。get()中的第二个参数指定了输出信号的 I/O 设备。

这些 FTP 指令在 Qt 的事件循环中排队并等待执行。QFtp 的 done(bool)信号表明这些指令的完成情况,构造函数中已经把这个信号与ftpDone(bool)连接起来了。

```
void FtpGet::ftpDone(bool error)
{
    if (error) {
        std::cerr << "Error: " << qPrintable(ftp.errorString())
                  << std::endl;
    } else {
        std::cerr << "File downloaded as "
                  << qPrintable(file.fileName()) << std::endl;
    }
    file.close();
    emit done();
}
```

这些 FTP 指令一旦得以执行,就可以关闭相应的文件并且发射我们自己的 done()信号。在此处关闭文件而不在 getFile()函数最后通过调用 ftp.close()来关闭文件,显得有些不同寻常;但是请牢记:在返回 getFile()函数之后,FTP 指令会异步执行并且整个执行过程会非常顺利。

QFtp 提供了一些 FTP 指令,包括 connectToHost()、login()、close()、list()、cd()、get()、put()、remove()、mkdir()、rmdir()和 rename()。所有这些函数都可以调度一个 FTP 指令,并且可以返回一个标识这个指令的 ID 号。对于传输模式(默认为被动模式)和传输类型(默认为二进制数据)的控制,也是可能的。

使用 rawCommand()可以执行任意的 FTP 指令。例如,以下是如何执行一个 SITE CHMOD 指令的代码:

```
ftp.rawCommand("SITE CHMOD 755 fortune");
```

当 QFtp 开始执行一个指令的时候,它发射 commandStarted(int)信号,而当这个指令完成的时候,它发射 commandFinished(int,bool)信号。int 参数是标识一个指令的 ID 号。如果对个别指令的结果感兴趣,则当调用这些指令的时候,可以保存这些 ID 号。了解并记录这些 ID 号可以为用户提供详细的反馈信息。例如:

```
bool FtpGet::getFile(const QUrl &url)
{
    ...
    connectId = ftp.connectToHost(url.host(), url.port(21));
    loginId = ftp.login();
    getId = ftp.get(url.path(), &file);
    closeId = ftp.close();
    return true;
}

void FtpGet::ftpCommandStarted(int id)
{
    if (id == connectId) {
```

```
        std::cerr << "Connecting..." << std::endl;
    } else if (id == loginId) {
        std::cerr << "Logging in..." << std::endl;
    ...
}
```

提供反馈的另一种方式是与 QFtp 的 stateChanged()信号连接,只要连接进入了一个新状态(QFtp::Connecting、QFtp::Connected、QFtp::LoggedIn,等等),就会发射 stateChanged()信号。

在绝大多数应用程序中,我们仅仅对整个指令序列的结果而不是对其中某个特别的指令感兴趣。这时,可以只简单地连接 done(bool)信号,一旦指令队列变空,就会发射这个信号。

当错误发生时,QFtp 会自动清空指令队列。也就是说,如果这次连接或者登录失败,位于队列后面的指令将不会执行。但如果在错误发生之后使用同一个 QFtp 对象调用新的指令序列,那么这些指令将会排队等待执行。

在应用程序的 .pro 文件中,需要如下的命令行来连接到 QtNetwork 数据库:

```
QT += network
```

现在来看一个更高级的例子。Spider 命令行程序下载一个 FTP 目录下的所有文件,它将从所有目录的子目录中递归下载这些文件。在 Spider 类中体现了网络的逻辑联系:

```
class Spider : public QObject
{
    Q_OBJECT

public:
    Spider(QObject *parent = 0);

    bool getDirectory(const QUrl &url);

signals:
    void done();

private slots:
    void ftpDone(bool error);
    void ftpListInfo(const QUrlInfo &urlInfo);

private:
    void processNextDirectory();

    QFtp ftp;
    QList<QFile *> openedFiles;
    QString currentDir;
    QString currentLocalDir;
    QStringList pendingDirs;
};
```

我们将开始目录指定为 QUrl,并且使用 getDirectory()来设置开始目录:

```
Spider::Spider(QObject *parent)
    : QObject(parent)
{
    connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
    connect(&ftp, SIGNAL(listInfo(const QUrlInfo &)),
            this, SLOT(ftpListInfo(const QUrlInfo &)));
}
```

在构造函数中,建立了两个信号-槽的连接。当为所获取的每一个文件请求一个目录列表[在 getDirectory()中]时,QFtp 就会发射 listInfo(const QUrlInfo &)信号。这个信号被连接到一个名为 ftpListInfo()的槽,它会下载与 URL 相关的给定文件。

```
bool Spider::getDirectory(const QUrl &url)
{
```

```
    if (!url.isValid()) {
        std::cerr << "Error: Invalid URL" << std::endl;
        return false;
    }

    if (url.scheme() != "ftp") {
        std::cerr << "Error: URL must start with 'ftp:'" << std::endl;
        return false;
    }

    ftp.connectToHost(url.host(), url.port(21));
    ftp.login();

    QString path = url.path();
    if (path.isEmpty())
        path = "/";

    pendingDirs.append(path);
    processNextDirectory();

    return true;
}
```

当调用 getDirectory()函数时,它首先要做一些检查,如果一切正常,就试图建立一个 FTP 连接。它对必须要处理的路径进行跟踪,并调用 processNextDirectory(),以开始下载根目录中的文件。

```
void Spider::processNextDirectory()
{
    if (!pendingDirs.isEmpty()) {
        currentDir = pendingDirs.takeFirst();
        currentLocalDir = "downloads/" + currentDir;
        QDir(".").mkpath(currentLocalDir);

        ftp.cd(currentDir);
        ftp.list();
    } else {
        emit done();
    }
}
```

processNextDirectory()函数从 pendingDirs 列表中取出第一个远程目录,同时在本地文件系统中创建一个相应的目录。然后它告诉 QFtp 对象将目录更改为被取出的目录并列出其中的文件。对于 list()处理的每一个文件,它都将发射一个促使 ftpListInfo()槽被调用的 listInfo()信号。

如果没有需要处理的目录了,该函数将发射 done()信号以表明下载完成。

```
void Spider::ftpListInfo(const QUrlInfo &urlInfo)
{
    if (urlInfo.isFile()) {
        if (urlInfo.isReadable()) {
            QFile *file = new QFile(currentLocalDir + "/"
                                    + urlInfo.name());

            if (!file->open(QIODevice::WriteOnly)) {
                std::cerr << "Warning: Cannot write file "
                          << qPrintable(QDir::toNativeSeparators(
                                        file->fileName()))
                          << ": " << qPrintable(file->errorString())
                          << std::endl;
                return;
            }

            ftp.get(urlInfo.name(), file);
            openedFiles.append(file);
        }
    } else if (urlInfo.isDir() && !urlInfo.isSymLink()) {
        pendingDirs.append(currentDir + "/" + urlInfo.name());
    }
}
```

ftpListInfo()槽的 urlInfo 参数提供了有关一个远程文件的详细信息。如果这个文件只是一个可读的普通文件(而不是目录),就调用 get()来下载它。这个用于下载的 QFile 对象是利用 new 函数以及一个指向它在 openedFiles 列表中存储的指针来分配的。

如果 QUrlInfo 持有一个不是符号连接的远程目录的细节信息,就将这个目录加到 pendingDirs 列表中。之所以要跳过符号连接,是因为它容易导致无穷递归循环。

```
void Spider::ftpDone(bool error)
{
    if (error) {
        std::cerr << "Error: " << qPrintable(ftp.errorString())
                  << std::endl;
    } else {
        std::cout << "Downloaded " << qPrintable(currentDir) << " to "
                  << qPrintable(QDir::toNativeSeparators(
                                QDir(currentLocalDir).canonicalPath()));
    }

    qDeleteAll(openedFiles);
    openedFiles.clear();

    processNextDirectory();
}
```

当所有这些 FTP 指令都完成后,或者如果有错误发生时,就会调用 ftpDone()槽。我们删除 QFile 对象以防止内存泄漏,同时也关闭每一个文件。最后,调用 processNextDirectory()。如果还有需要处理的目录,整个过程将从列表中的下一个目录重新开始;否则,下载过程停止并且发射 done()信号。

如果没有错误发生,FTP 指令和信号的序列如下:

```
connectToHost(host, port)
login()

cd(directory_1)
list()
    emit listInfo(file_1_1)
        get(file_1_1)
    emit listInfo(file_1_2)
        get(file_1_2)
    ...
emit done()

...

cd(directory_N)
list()
    emit listInfo(file_N_1)
        get(file_N_1)
    emit listInfo(file_N_2)
        get(file_N_2)
    ...
emit done()
```

如果打开的文件实际上是一个目录,它将被加入到 pendingDirs 列表中。在当前 list()指令中的最后一个文件被下载完时,就将发出一个新的 cd()指令以及下一个待处理目录的 list()指令,而整个过程将随一个新目录重新开始。这个过程不断重复,既有新文件被下载也有新目录被添加到 pendingDirs 列表,直到每一目录下的文件都被下载完,这时 pendingDirs 列表也最终将变空。

如果要下载一个目录下的 20 个文件,当下载到第 5 个文件的时候,网络发生错误,那么剩下的文件将不会被下载。如果想下载尽可能多的文件,一个解决方案是一次只调用一个 GET 操作来

下载一个文件,并且在调用下一个新的 GET 操作之前等待 done(bool)信号。在 listInfo()中,将简单地把文件名追加到一个 QStringList 中,而不是立即调用 get(),而且在 done(bool)中,我们将对下一个文件调用 get()函数以在 QStringList 中下载。于是,整个执行顺序看起来可以归结如下:

```
connectToHost(host, port)
login()
cd(directory_1)
list()
...
cd(directory_N)
list()
    emit listInfo(file_1_1)
    emit listInfo(file_1_2)
    ...
    emit listInfo(file_N_1)
    emit listInfo(file_N_2)
    ...
emit done()
get(file_1_1)
emit done()
get(file_1_2)
emit done()
...
get(file_N_1)
emit done()
get(file_N_2)
emit done()
...
```

另外一个解决方案是对每一个文件使用一个 QFtp 对象。这样可以通过一些单独的 FTP 连接来并行下载这些文件。

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QStringList args = QCoreApplication::arguments();
    if (args.count() != 2) {
        std::cerr << "Usage: spider url" << std::endl
                  << "Example:" << std::endl
                  << "    spider ftp://ftp.trolltech.com/freebies/"
                  << "leafnode" << std::endl;
        return 1;
    }

    Spider spider;
    if (!spider.getDirectory(QUrl(args[1])))
        return 1;

    QObject::connect(&spider, SIGNAL(done()), &app, SLOT(quit()));

    return app.exec();
}
```

main()函数完成了这个程序。如果用户没有在命令行中指定一个 URL,就给出一个出错信息并终止程序。

在这两个 FTP 实例中,利用 get()函数获得的数据都被写入 QFile 中。但这样的做法并不是必需的。如果想把这些数据放到内存中,则可以使用 QBuffer,它是一个在 QByteArray 上操作的 QIODevice 的子类。例如:

```
QBuffer *buffer = new QBuffer;
buffer->open(QIODevice::WriteOnly);
ftp.get(urlInfo.name(), buffer);
```

也可以省略 get()中的输入/输出(I/O)设备参数,或者只传递一个空指针。然后每次当有新数据可读的时候,QFtp 类就会发射一个 readyRead()信号,并且可以使用 read()或者 readAll()来读取这些数据。

15.2 写 HTTP 客户端

QHttp 类在 Qt 中实现了 HTTP 协议的客户端程序。它提供了各种各样的函数来执行绝大多数普通 HTTP 操作,包括 get()和 post(),并且还提供了一个发送任意 HTTP 请求指令的方式。如果已经阅读过前面有关 QFtp 的章节,那么将会发现 QFtp 和 QHttp 之间存在很多相似之处。

QHttp 类是异步工作的。当调用一个像 get()或者 post()这样的函数时,它会立即返回,并且当控制权回到 Qt 事件循环时才会开始传输数据。这样就确保了在处理 HTTP 请求时,应用程序的用户界面可以始终保持响应。

我们将查看一个名为 httpget 的控制台应用程序,以说明如何利用 HTTP 协议下载一个文件。这里将不再显示出头文件,因为它和前一节所使用的那个 ftpget 例子,不论在功能上还是实现过程上,都非常相似。

```
HttpGet::HttpGet(QObject *parent)
    : QObject(parent)
{
    connect(&http, SIGNAL(done(bool)), this, SLOT(httpDone(bool)));
}
```

在构造函数中,我们把 QHttp 对象的 done(bool)信号与 httpDone(bool)私有槽连接起来。

```
bool HttpGet::getFile(const QUrl &url)
{
    if (!url.isValid()) {
        std::cerr << "Error: Invalid URL" << std::endl;
        return false;
    }

    if (url.scheme() != "http") {
        std::cerr << "Error: URL must start with 'http:'" << std::endl;
        return false;
    }

    if (url.path().isEmpty()) {
        std::cerr << "Error: URL has no path" << std::endl;
        return false;
    }

    QString localFileName = QFileInfo(url.path()).fileName();
    if (localFileName.isEmpty())
        localFileName = "httpget.out";

    file.setFileName(localFileName);
    if (!file.open(QIODevice::WriteOnly)) {
        std::cerr << "Error: Cannot write file "
                  << qPrintable(file.fileName()) << ": "
                  << qPrintable(file.errorString()) << std::endl;
        return false;
    }

    http.setHost(url.host(), url.port(80));
    http.get(url.path(), &file);
    http.close();
    return true;
}
```

getFile()函数执行与之前所给出的 FtpGet::getFile()一样可以执行同种类型的错误检查,并且采用相同的方式来命名文件的本地文件名。当从网站获得文件时,由于不必登录,所以只要设置主机和端口号(如果在 URL 中没有指定端口号,则采用默认的 HTTP 端口号 80),就可将数据下载到文件中,因为 QHttp::get()的第二个参数指定了输出信号的输入输出(I/O)设备。

这些 HTTP 请求在 Qt 的事件循环中排队并且被异步执行。QHttp 的 done(bool)信号表明了这些请求的完成情况,在构造函数中已经把这个信号与 httpDone(bool)连接起来了。

```
void HttpGet::httpDone(bool error)
{
    if (error) {
        std::cerr << "Error: " << qPrintable(http.errorString())
                  << std::endl;
    } else {
        std::cerr << "File downloaded as "
                  << qPrintable(file.fileName()) << std::endl;
    }
    file.close();
    emit done();
}
```

一旦完成了这些 HTTP 请求,就关闭这个文件,并在有错误发生时通知用户。

它的 main()函数与 ftpget 中曾使用过的主函数非常相似:

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QStringList args = QCoreApplication::arguments();

    if (args.count() != 2) {
        std::cerr << "Usage: httpget url" << std::endl
                  << "Example:" << std::endl
                  << "    httpget http://doc.trolltech.com/index.html"
                  << std::endl;
        return 1;
    }

    HttpGet getter;
    if (!getter.getFile(QUrl(args[1])))
        return 1;

    QObject::connect(&getter, SIGNAL(done()), &app, SLOT(quit()));

    return app.exec();
}
```

QHttp 类提供了多种操作,包括 setHost()、get()、post()和 head()。如果站点需要认证,则 setUser()可以用来提供用户名和口令。QHttp 可以用程序员自编的套接字装置而不用其内部自带的 QTcpSocket。这就使利用可靠的 QtSslSocket 来在 SSL(加密套接字协议层)或 TLS(加密传输协议层)上实现 HTTP 成为可能。

为了向 CGI 脚本发送一列"name = value"值对,可以使用 post()函数:

```
http.setHost("www.example.com");
http.post("/cgi/somescript.py", "x=200&y=320", &file);
```

既可以用 8 位的字符串来传递数据,也可以像 QFile 一样,通过传递一个开放的 QIODevice 来传递数据。为了获得更多的控制权,可以使用 request()函数,它接收任意一个 HTTP 的标题和数据。例如:

```
QHttpRequestHeader header("POST", "/search.html");
header.setValue("Host", "www.trolltech.com");
header.setContentType("application/x-www-form-urlencoded");
http.setHost("www.trolltech.com");
http.request(header, "qt-interest=on&search=opengl");
```

当 QHttp 开始执行请求时，它会发射 requestStarted(int)信号，而当这个请求完成时，会发射 requestFinished(int,bool)信号。int 参数是标识请求的 ID 号。如果我们对个别请求的结果感兴趣，就可以在调用这些指令的时候保存 ID 号。了解并记录这些 ID 号可以为用户提供详细的反馈信息。

在绝大多数应用程序中，我们仅仅想知道整个系列的请求是否已成功地完成执行。通过与 done(bool)信号连接，就可以很简单地实现这一点，当请求序列变空时，就会发射该信号。

当有错误发生时，这个请求队列会被自动清空。但是如果在错误发生之后使用相同的 QHttp 对象调用新的请求，这些请求将会照常排队并被发送执行。

与 QFtp 一样，QHttp 不仅提供了 read()和 readAll()函数，它还提供 readyRead()信号，使我们不必指定某一输入/输出设备。

15.3 写 TCP 客户/服务器应用程序

QTcpSocket 和 QTcpServer 类可以用来实现 TCP 客户端和服务器。TCP 是一个传输协议，它构成了包括 FTP 和 HTTP 等很多应用程序层的因特网协议基础，它也可以用于定制用户自己的协议。

TCP 是一个基于流的协议。对于应用程序，数据表现为一个长长的流，而不是一个大的平面文件。在 TCP 之上建立的高层协议通常是基于行的或者基于块的。

- 基于行的协议把数据作为一行文本进行传输，每一数据行都以一个换行符结尾。
- 基于块的协议把数据作为二进制块进行传输。每一数据块都是由一个大小字段及其包含的数据组成的。

QTcpSocket 间接地由 QIODevice 派生而来(通过 QAbstractSocket)，所以它可以使用 QDataStream 或者 QTextStream 来进行读取和写入。与从文件中读取数据相比，从网络上读取数据会有一个值得注意的差别：在使用 >> 操作符之前，必须确定已经从另一端接收了足够多的数据。如果没有接收到足够多的数据就使用 >> 操作符，通常会导致不确定的状态发生。

这一节将查看一个客户端和服务器应用程序的代码，而该客户端和服务器应用程序使用了自定义基于块的协议。如图 15.1 所示的客户端称为 Trip Planner，它允许用户做出下一次乘坐火车的旅行计划。而服务器称为 Trip Server，它向客户端提供旅行信息。我们将从编写这个 Trip Planner 客户端应用程序开始。

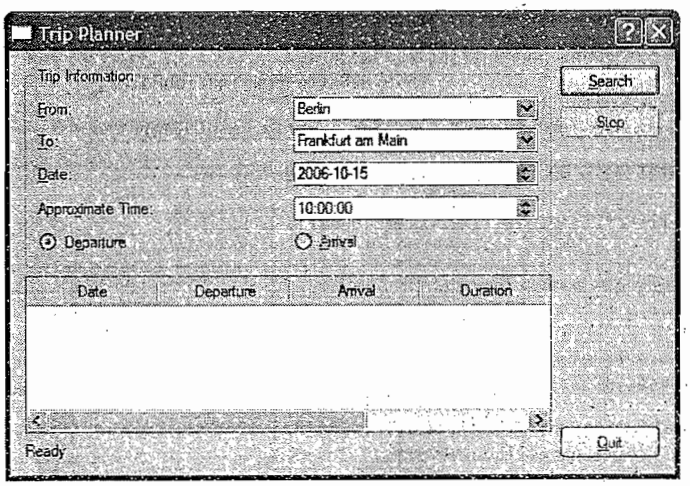


图 15.1 Trip Planner 应用程序

Trip Planner 提供了一个 From 字段、一个 To 字段、一个 Date 字段、一个 Approximate Time 字段以及两个单选钮来选择火车出发或者到达的大概时间。当用户单击 Search 按钮,应用程序会向服务器发送一个请求,服务器返回一个匹配这个用户要求的火车旅行列表。这个列表被显示在 Trip Planner 窗口的 QTableWidget 中。窗口的底部是显示最后一次操作状态的 QLabel 和一个 QProgressBar。

利用 Qt 设计师,Trip Planner 的用户界面被创建于一个名为 tripplanner.ui 的文件中。这里,我们会把注意力集中在实现应用程序功能的 QDialog 子类的源代码上:

```
#include "ui_tripplanner.h"

class QPushButton;

class TripPlanner : public QDialog, private Ui::TripPlanner
{
    Q_OBJECT

public:
    TripPlanner(QWidget *parent = 0);

private slots:
    void connectToServer();
    void sendRequest();
    void updateTableWidget();
    void stopSearch();
    void connectionClosedByServer();
    void error();

private:
    void closeConnection();

    QPushButton *searchButton;
    QPushButton *stopButton;
    QTcpSocket tcpSocket;
    quint16 nextBlockSize;
};
```

除了 QDialog 之外,TripPlanner 类也派生自 Ui::TripPlanner(它是通过 uic 从 tripplanner.ui 中生成的)。tcpSocket 成员变量封装了 TCP 连接。在解析从服务器接收的数据块时,则会使用到 nextBlockSize 变量。

```
TripPlanner::TripPlanner(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);

    searchButton = buttonBox->addButton(tr("&Search"),
                                        QDialogButtonBox::ActionRole);
    stopButton = buttonBox->addButton(tr("S&top"),
                                      QDialogButtonBox::ActionRole);
    stopButton->setEnabled(false);
    buttonBox->button(QDialogButtonBox::Close)->setText(tr("&Quit"));

    QDateTime dateTime = QDateTime::currentDateTime();
    dateEdit->setDate(dateTime.date());
    timeEdit->setTime(QTime(dateTime.time().hour(), 0));

    progressBar->hide();
    progressBar->setSizePolicy(QSizePolicy::Preferred,
                               QSizePolicy::Ignored);

    tableWidget->verticalHeader()->hide();
    tableWidget->setEditTriggers(QAbstractItemView::NoEditTriggers);

    connect(searchButton, SIGNAL(clicked()),
            this, SLOT(connectToServer()));
```

```
    connect(stopButton, SIGNAL(clicked()), this, SLOT(stopSearch()));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));

    connect(&tcpSocket, SIGNAL(connected()), this, SLOT(sendRequest()));
    connect(&tcpSocket, SIGNAL(disconnected()),
            this, SLOT(connectionClosedByServer()));
    connect(&tcpSocket, SIGNAL(readyRead()),
            this, SLOT(updateTableWidget()));
    connect(&tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),
            this, SLOT(error()));
}
```

在构造函数中,我们基于当前的日期和时间来初始化日期和时间编辑器。我们也会隐藏进度条,因为只想在连接被激活时才显示它。在 Qt 设计师中,滚动条的 minimum 和 maximum 属性都被设置为 0,这就意味着 QProgressBar 会表现得像一个忙碌的指示器而不是一个标准的基于百分比的进度条。

在构造函数中,还把 QTcpSocket 的 connected()、disconnected()、readyRead()和 error(QAbstractSocket::SocketError)信号与私有槽连接起来。

```
void TripPlanner::connectToServer()
{
    tcpSocket.connectToHost("tripserver.zugbahn.de", 6178);

    tableWidget->setRowCount(0);
    searchButton->setEnabled(false);
    stopButton->setEnabled(true);
    statusLabel->setText(tr("Connecting to server..."));
    progressBar->show();

    nextBlockSize = 0;
}
```

当用户单击 Search 按钮开始搜索的时候,就会执行 connectToServer()槽。我们在 QTcpSocket 对象上调用 connectToHost()从而连接到服务器,这里假设在 Tripserver.zugbahn.de 虚拟主机上的 6178 端口是可以访问的。(如果想在自己的机器上尝试这个实例,请把主机名称替换为 QHostAddress::LocalHost)。connectToHost()调用是异步的,它总是立即返回。连接通常会在稍后建立。当连接建立起来并运行时,QTcpSocket 对象发射 connected()信号;如果连接失败,QTcpSocket 对象会发射 error(QAbstractSocket::SocketError)信号。

接下来,更新用户界面并且将进度条设置为可见。

最后,将 nextBlockSize 变量设置为 0。这个变量用于存储从服务器所接收的下一个块的长度值。此处选择的是 0 值,这意味着还不知道下一个块的大小。

```
void TripPlanner::sendRequest()
{
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_3);
    out << quint16(0) << quint8('S') << fromComboBox->currentText()
        << toComboBox->currentText() << dateEdit->date()
        << timeEdit->time();

    if (departureRadioButton->isChecked()) {
        out << quint8('D');
    } else {
        out << quint8('A');
    }
    out.device()->seek(0);
    out << quint16(block.size() - sizeof(quint16));
    tcpSocket.write(block);

    statusLabel->setText(tr("Sending request..."));
}
```

当 QTcpSocket 对象发射 connected()信号时,会执行 sendRequest()槽,表明一个连接已经被建立。这个槽的任务是向服务器生成一个请求,其中包含用户输入的所有信息。

这个请求是如下格式的二进制数据块:

quint16	字节数据中块的大小(不包括这个字段)
quint8	请求指令类型(总是为'S')
QString	出发城市名
QString	到站城市名
QDate	发车日期
QTime	火车大致运行时间
quint8	火车发车时间('D')或到站时间('A')

首先把这些数据写到一个称为 block 的 QByteArray 中。不能直接把数据写到 QTcpSocket 中,因为在把所有数据都放到这个数据块里以前,我们并不知道块的大小,而块的大小必须先发送出去。

我们一开始写入 0 值作为块的大小,以及其他数据。然后,对输入/输出设备(在后台是由 QDataStream 创建的 QBuffer)调用 seek(0)以重新移动到字节数组的开始处,同时利用数据块的实际尺寸值覆盖最初写入的 0 值。这个尺寸值是通过由数据块的尺寸减去 sizeof(quint16)(即 2)计算得到的,也就是去掉前面容量字段所占用的空间。在这之后,对 QTcpSocket 调用 write()向服务器发送这个块。

```
void TripPlanner::updateTableWidget()
{
    QDataStream in(&tcpSocket);
    in.setVersion(QDataStream::Qt_4_3);

    forever {
        int row = tableWidget->rowCount();

        if (nextBlockSize == 0) {
            if (tcpSocket.bytesAvailable() < sizeof(quint16))
                break;
            in >> nextBlockSize;
        }

        if (nextBlockSize == 0xFFFF) {
            closeConnection();
            statusLabel->setText(tr("Found %1 trip(s)").arg(row));
            break;
        }

        if (tcpSocket.bytesAvailable() < nextBlockSize)
            break;

        QDate date;
        QTime departureTime;
        QTime arrivalTime;
        quint16 duration;
        quint8 changes;
        QString trainType;

        in >> date >> departureTime >> duration >> changes >> trainType;
        arrivalTime = departureTime.addSecs(duration * 60);
        tableWidget->setRowCount(row + 1);

        QStringList fields;
        fields << date.toString(Qt::LocalDate)
               << departureTime.toString(tr("hh:mm"))
               << arrivalTime.toString(tr("hh:mm"))
```

```
                << tr("%1 hr %2 min").arg(duration / 60)
                                     .arg(duration % 60)
                << QString::number(changes)
                << trainType;
        for (int i = 0; i < fields.count(); ++i)
            tableWidget->setItem(row, i,
                                 new QTableWidgetItem(fields[i]));
        nextBlockSize = 0;
    }
}
```

updateTableWidget()槽被连接到 QTcpSocket 的 readyRead()信号，只要 QTcpSocket 已经从服务器收到新数据，就会发射该信号。服务器向我们发送一个和这个用户要求匹配的可能的火车旅行列表。每一个匹配的旅行都作为一个单独的块发送，并且每一个块的开始都是块的大小尺寸值。图 15.2 举例说明了这类数据块流。因为我们不需要每次从服务器得到一整个块的数据，所以 forever 循环是非常必要的[①]。我们也许只是收到一个完整的块、一个块的一部分、一个块和一个块的一部分，甚至还可能一次收到所有的块数据。

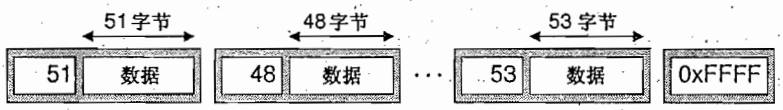


图 15.2　Trip 服务器的数据块

forever 循环是如何工作的呢？如果 nextBlockSize 变量为 0，则意味着还没有读取到下一个块的大小。我们尝试去读取它(假设至少已经有两个字节读取)。服务器使用一个大小为 0xFFFF 的值来表示没有更多的数据可以接收，所以如果读取到该值，就表明已经读取到数据块的末尾了。

如果块的大小不是 0xFFFF，则尝试去下一个块中读取。首先，我们查看是否有块的容量大小这么多字节可以读取；如果没有，就先在这里停止。当有更多数据可以读取的时候，readyRead()信号将会被再次发射，然后就可以再次尝试读取。

一旦确认一个完整的块已经被读取到，就可以在 QDataStream 上安全地使用 >> 操作符以提取与一个旅行相关的信息，并且可以使用这些信息来创建 QTableWidgetItems。从服务器上接收的块具有如下格式：

quint16	字节数据中块的大小(不包括这个字段)
QDate	出发日期
QTime	出发时间
quint16	持续时间 (以分钟计)
quint8	换乘次数
QString	火车类型

最后，重新将 nextBlockSize 变量设置为 0，以表示下一个块的大小是未知的并且需要读取。

```
void TripPlanner::closeConnection()
{
    tcpSocket.close();
    searchButton->setEnabled(true);
    stopButton->setEnabled(false);
    progressBar->hide();
}
```

① forever 关键字是 Qt 自带的，它可以简单地展开为 for (;;)。

closeConnection()私有函数关闭到 TCP 服务器的连接,并且更新了用户界面。当读到 0xFFFF 时,它会被 updateTableWidget()调用,并且在稍候要介绍的几个槽中,它也会被调用到。

```
void TripPlanner::stopSearch()
{
    statusLabel->setText(tr("Search stopped"));
    closeConnection();
}
```

stopSearch()槽被连接到 Stop 按钮的 clicked()信号。从本质上来说,它只不过是调用 closeConnection()而已。

```
void TripPlanner::connectionClosedByServer()
{
    if (nextBlockSize != 0xFFFF)
        statusLabel->setText(tr("Error: Connection closed by server"));
    closeConnection();
}
```

connectionClosedByServer()槽被连接到 QTcpSocket 的 disconnected()信号。如果服务器关闭这个连接且我们还没有收到 0xFFFF 数据终止符,就告诉用户有一个错误发生了。我们会像往常一样调用 closeConnection()来更新用户界面。

```
void TripPlanner::error()
{
    statusLabel->setText(tcpSocket.errorString());
    closeConnection();
}
```

error()槽被连接到 QTcpSocket 的 error(QAbstractSocket::SocketError)信号。我们忽略错误代码而使用 QTcpSocket::errorString(),它将为最后一次发生的错误返回一个用户可读的出错信息。

所有这些都是为了 TripPlanner 类。Trip Planner 应用程序的 main()函数正如我们派生自预期的那样:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TripPlanner tripPlanner;
    tripPlanner.show();
    return app.exec();
}
```

现在来实现服务器。服务器包含两个类:TripServer 和 ClientSocket。TripServer 类派生自 QTcpServer,这是一个允许接收来访的 TCP 连接的类。ClientSocket 重新实现了 QTcpSocket,处理一个单独的连接。在任何时候,在内存中 ClientSocket 对象的数量和正在被服务的客户端数量都是一样多的。

```
class TripServer : public QTcpServer
{
    Q_OBJECT

public:
    TripServer(QObject *parent = 0);

private:
    void incomingConnection(int socketId);
};
```

tripServer 类通过 QTcpServer 重新实现了 incomingConnection()函数。只要有一个客户端试图连接到服务器正监听的端口,这个函数就会被调用。

```
TripServer::TripServer(QObject *parent)
    : QTcpServer(parent)
{
}
```

tripServer构造函数则非常普通。

```
void TripServer::incomingConnection(int socketId)
{
    ClientSocket *socket = new ClientSocket(this);
    socket->setSocketDescriptor(socketId);
}
```

在incomingConnection()中,创建了一个ClientSocket对象作为tripServer对象的子对象,并且将它的套接字描述符设置成提供给我们的数字。当连接终止时,ClientSocket对象将自动删除。

```
class ClientSocket : public QTcpSocket
{
    Q_OBJECT

public:
    ClientSocket(QObject *parent = 0);

private slots:
    void readClient();

private:
    void generateRandomTrip(const QString &from, const QString &to,
                            const QDate &date, const QTime &time);

    quint16 nextBlockSize;
};
```

ClientSocket类派生自QTcpSocket并且封装了一个单独客户端的状态。

```
ClientSocket::ClientSocket(QObject *parent)
    : QTcpSocket(parent)
{
    connect(this, SIGNAL(readyRead()), this, SLOT(readClient()));
    connect(this, SIGNAL(disconnected()), this, SLOT(deleteLater()));

    nextBlockSize = 0;
}
```

在构造函数中,我们建立了必要的信号-槽的连接,并且将nextBlockSize变量设置为0,这表示还不知道由客户端发送的块的大小。

disconnected()信号被连接到deleteLater(),这是一个从QObject继承的函数,当控制权返回到Qt的事件循环时,它会删除对象。这样就确保当关闭套接字连接时,ClientSocket对象会被删除。

```
void ClientSocket::readClient()
{
    QDataStream in(this);
    in.setVersion(QDataStream::Qt_4_3);

    if (nextBlockSize == 0) {
        if (bytesAvailable() < sizeof(quint16))
            return;
        in >> nextBlockSize;
    }
    if (bytesAvailable() < nextBlockSize)
        return;

    quint8 requestType;
    QString from;
    QString to;
    QDate date;
    QTime time;
    quint8 flag;

    in >> requestType;
    if (requestType == 'S') {
        in >> from >> to >> date >> time >> flag;
```

```
        std::srand(from.length() * 3600 + to.length() * 60
                + time.hour());
        int numTrips = std::rand() % 8;
        for (int i = 0; i < numTrips; ++i)
            generateRandomTrip(from, to, date, time);

        QDataStream out(this);
        out << quint16(0xFFFF);
    }

    close();
}
```

readClient()槽被连接到 QTcpSocket 的 readyRead()信号。如果 nextBlockSize 为 0,就首先读取块的大小;否则,就表明已经开始读取它了,并且还要检查是否读取一个完整的块的时机已经到来。一旦一个完整的块已经为读取做好准备,就一次读取它。我们直接在 QTcpSocket(this 对象)上使用 QDataStream,并且利用 >> 操作符来读取各个字段。

一旦读取了客户端的请求,就生成一个应答。如果这是一个真正的应用程序,我们就会在一个火车时刻表数据库中查找信息并且试图找到相匹配的火车车次,但是这里我们使用一个称为 generateRandomTrip()的函数来生成一次随机的旅行就够了。我们任意次地调用这个函数,然后发送 0xFFFF 表示这个数据的结束。最后,关闭连接。

```
void ClientSocket::generateRandomTrip(const QString & /* from */,
        const QString & /* to */, const QDate &date, const QTime &time)
{
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_3);
    quint16 duration = std::rand() % 200;
    out << quint16(0) << date << time << duration << quint8(1)
        << QString("InterCity");
    out.device()->seek(0);
    out << quint16(block.size() - sizeof(quint16));

    write(block);
}
```

generateRandomTrip()函数展示了如何在一个 TCP 连接之上发送一个数据块。这与客户端程序的 sendRequest()函数中的做法(见 281 页)非常相似。我们再一次把这个块写入到 QByteArray,这样就可以在使用 write()发送数据之前知道它的大小了。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TripServer server;
    if (!server.listen(QHostAddress::Any, 6178)) {
        std::cerr << "Failed to bind to port" << std::endl;
        return 1;
    }

    QPushButton quitButton(QObject::tr("&Quit"));
    quitButton.setWindowTitle(QObject::tr("Trip Server"));
    QObject::connect(&quitButton, SIGNAL(clicked()),
                     &app, SLOT(quit()));
    quitButton.show();
    return app.exec();
}
```

在 main()中,我们创建了 tripServer 对象和允许用户停止服务器的 QPushButton。我们通过调用 QTcpSocket::listen()来启动服务器,它将具有我们想接收的连接的 IP 地址和端口号。专门的地址 0.0.0.0 (QHostAddress::Any)表示在本地主机上的任意的 IP 接口。

在程序开发中，使用 QPushButton 来代表服务器是非常方便的。然而，一个配备过的服务器应该运行于没有图形用户界面的情况下，就像 Windows 服务或者 UNIX 的端口监控程序一样。为此，Trolltech 公司提供了一个名为 QtService 的商用扩展软件来辅助实现这一功能。

现在就完成了这个客户/服务器实例。在这个实例中，我们使用了一个基于块的协议，它允许使用 QDataStream 来读取和写入。如果想使用基于行的协议，最简单的方式是在一个连接到 readyRead()信号的槽中使用 QTcpSocket 的 canReadLine()和 readLine()函数：

```
QStringList lines;
while (tcpSocket.canReadLine())
    lines.append(tcpSocket.readLine());
```

然后，处理已经读取的每一行。至于发送数据，可以通过在 QTcpSocket 上使用 QTextStream 来完成。

这里使用的服务器实现，在同时有较多连接的时候，不能很好地并行工作。其原因在于：当处理某一个请求时，并没有同时处理其他连接。一个更好的方式是为每一个连接启动一个新的线程。位于 Qt 中 examples/network/threadedfortuneserver 目录下的例子 Threaded Fortune Server，将会给出这种方法的具体操作步骤。

15.4　发送和接收 UDP 数据报

QUdpSocket 类可以用来发送和接收 UDP 数据报(datagram)。UDP 是一种不可靠的，面向数据报的协议。一些应用层的协议使用 UDP，因为它比 TCP 更加小巧轻便。采用 UDP，数据是以包(数据报)的形式从一个主机发送到另一个主机的。这里并没有连接的概念，而且如果 UDP 包没有被成功投递，它不会向发送者报告任何错误。

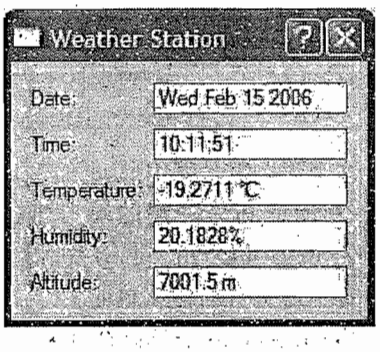


图 15.3　Weather Station 应用程序

我们将会通过 Weather Balloon 和 Weather Station 这两个实例来看看在 Qt 应用程序中是如何使用 UDP 的。Weather Balloon 应用程序模拟气象气球的功能，每 2 秒钟就发送一个包含当前天气情况的 UDP 数据报(假设使用无线连接)。Weather Station 应用程序(如图 15.3 所示)接收这些数据报并且使这些信息显示在屏幕上。首先查看 Weather Balloon 的代码：

```
class WeatherBalloon : public QPushButton
{
    Q_OBJECT

public:
    WeatherBalloon(QWidget *parent = 0);

    double temperature() const;
    double humidity() const;
    double altitude() const;

private slots:
    void sendDatagram();

private:
    QUdpSocket udpSocket;
    QTimer timer;
};
```

WeatherBalloon类派生自 QPushButton。它借助 QPushButton 的 QUdpSocket 私有变量与 Weather Station 进行通信。

```
WeatherBalloon::WeatherBalloon(QWidget *parent)
    : QPushButton(tr("Quit"), parent)
{
    connect(this, SIGNAL(clicked()), this, SLOT(close()));
    connect(&timer, SIGNAL(timeout()), this, SLOT(sendDatagram()));

    timer.start(2 * 1000);
    setWindowTitle(tr("Weather Balloon"));
}
```

在构造函数中,利用一个 QTimer 来实现每 2 秒钟调用一次 sendDatagram()。

```
void WeatherBalloon::sendDatagram()
{
    QByteArray datagram;
    QDataStream out(&datagram, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_3);
    out << QDateTime::currentDateTime() << temperature() << humidity()
        << altitude();

    udpSocket.writeDatagram(datagram, QHostAddress::LocalHost, 5824);
}
```

在 sendDatagram()中,生成并发送一个包含当前日期、时间、温度、湿度和高度的数据报:

QDateTime	测量的日期及时间
double	温度(℃)
double	湿度(%)
double	高度(m)

这个数据报是利用 QUdpSocket::writeDatagram()发送的。writeDatagram()的第二个和第三个参数是 IP 地址和另一端的端口号(Weather Station)。对于这个实例,我们假设 Weather Station 和 Weather Balloon 运行在同一台机器上,所以使用 127.0.0.1(QHostAddress::LocalHost)的 IP 地址,它是指定本地主机的特殊地址。

与 QTcpSocket::connectToHost()不同,QUdpSocket::writeDatagram()不接受主机名称,而只能使用主机地址。如果想在这里把主机名称解析成为它的 IP 地址,有两种选择:在查找发生时阻塞,然后使用 QHostInfo::fromName()静态函数;或者,使用 QHostInfo::lookupHost()静态函数。当查找完成时,它将立即返回,同时利用含有相应地址的 QHostInfo 对象传递而调用槽。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    WeatherBalloon balloon;
    balloon.show();
    return app.exec();
}
```

main()函数仅仅创建了一个 WeatherBalloon 对象,它既作为一个 UDP 端进行服务,也作为一个 QPushButton 显示在屏幕上。单击 QPushButton,就可以退出应用程序。

现在看一下 Weather Station 客户端的源代码:

```
class WeatherStation : public QDialog
{
    Q_OBJECT

public:
```

```
    WeatherStation(QWidget *parent = 0);

private slots:
    void processPendingDatagrams();

private:
    QUdpSocket udpSocket;

    QLabel *dateLabel;
    QLabel *timeLabel;
    ...
    QLineEdit *altitudeLineEdit;
};
```

WeatherStation 类派生自 QDialog。它监听一个特定的 UDP 端口,解析任何到来的数据报(来自于 Weather Balloon),并且把它们的内容显示到 5 个只读的 QLineEdit 中。这里唯一需要注意的私有变量是 QUdpSocket 类型的 udpSocket,我们将利用它来接收数据报。

```
WeatherStation::WeatherStation(QWidget *parent)
    : QDialog(parent)
{
    udpSocket.bind(5824);

    connect(&udpSocket, SIGNAL(readyRead()),
            this, SLOT(processPendingDatagrams()));
    ...
}
```

在构造函数中,首先把 QUdpSocket 绑定到 Weather Balloon 所传送的端口。因为我们没有指定主机地址,套接字将在运行 Weather Station 的机器上接收发往任意 IP 地址的数据报。然后,将套接字的 readyRead()信号与提取和显示数据的 processPendingDatagrams()私有槽连接起来。

```
void WeatherStation::processPendingDatagrams()
{
    QByteArray datagram;

    do {
        datagram.resize(udpSocket.pendingDatagramSize());
        udpSocket.readDatagram(datagram.data(), datagram.size());
    } while (udpSocket.hasPendingDatagrams());

    QDateTime dateTime;
    double temperature;
    double humidity;
    double altitude;

    QDataStream in(&datagram, QIODevice::ReadOnly);
    in.setVersion(QDataStream::Qt_4_3);
    in >> dateTime >> temperature >> humidity >> altitude;

    dateLineEdit->setText(dateTime.date().toString());
    timeLineEdit->setText(dateTime.time().toString());
    temperatureLineEdit->setText(tr("%1 °C").arg(temperature));
    humidityLineEdit->setText(tr("%1%").arg(humidity));
    altitudeLineEdit->setText(tr("%1 m").arg(altitude));
}
```

当接收到数据报时,就调用 processPendingDatagrams()槽。QUdpSocket 将收到的数据报进行排队并让我们可以一次一个地读取它们。通常情况下,应该只有一个数据报,但是不能排除在发射 readyRead()信号前发送端连续发送一些数据报的可能性。如果那样的话,可以忽略除最后一个以外的其他所有数据报,因为之前的数据报包含的只是过期的天气情况。

pendingDatagramSize()函数返回第一个待处理的数据报的大小。从应用程序的角度来看,数据报总是作为一个单一的数据单元来发送和接收的。这意味着只要有任意字节的数据可用,就认为整个数据报都可以被读取。readDatagram()调用把第一个待处理的数据报的内容复制到指定的

char * 缓冲区中(如果缓冲区空间太小,就直接截断数据),并且前移至下一个待处理的数据报。一旦读取了所有的数据报,就把最后一个数据报(包含最新气象状况测量值的数据报)分解为几个部分,并且用新的数据来组装 QLineEdit。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    WeatherStation station;
    station.show();
    return app.exec();
}
```

最后,在 main()中,我们创建并显示 WeatherStation。

至此就完成了 UDP 发送端和接收端的代码。这个应用程序已经尽可能地简单了,它只是由 Weather Balloon 发送数据报,由 Weather Station 接收它们。在绝大多数实际应用中,这两个应用程序都需要通过它们的套接字读取和写入。QUdpSocket::writeDatagram()函数可以传递主机地址和端口号,所以 QUdpSocket 可以从用 bind()绑定的主机和端口读取数据,并且将其写入到其他的主机和端口。

第 16 章　XML

XML(eXtensible Markup Language,可扩展标记语言)是普遍用于数据交换和数据存储的一种多用途文本文件格式。XML 首先是由万维网协会(World Wide Web Consortium,W3C)作为 SGML 的一个替代品来开发的。它的语法规则与 HTML 相似,不过 XML 是一种用于语言分析的语言,它并没有要求专门的标记符、属性或者条目。HTML 的 XML 兼容版称为 XHTML。

对于比较流行的 SVG(可标量化矢量图形)XML 格式,QtSvg 模块提供了可用于载入并呈现 SVG 图像的类。对于使用 MathML(数学标记语言)XML 格式的绘制文档,可以使用 Qt Solutions 中的 QtMmlWidget。

对于一般的 XML 数据处理,Qt 提供了 QtXml 模块,这是本章的主题[①]。QtXml 模块提供了三种截然不同的应用程序编程接口用来读取 XML 文档:

- QXmlSreamReader 是一个用于读取格式良好的 XML 文档的快速解析器。
- DOM(文档对象模型) 把 XML 文档转换为应用程序可以遍历的树形结构。
- SAX(XML 简单应用程序编程接口) 通过虚拟函数直接向应用程序报告"解析事件"。

QXmlStreamReader 类最快且最易于使用,它同时还提供了与其他 Qt 兼容的应用程序编程接口。它很适用于编写单通解析器。DOM 的主要优点是它能以任意顺序遍历 XML 文档的树形表示,同时可以实现多通解析算法。有一些应用程序甚至使用 DOM 树作为它们的基本数据结构。SAX 则因为一些历史原因而被得以沿用至今,使用 QXmlStreamReader 通常会有更加简单高效的编码。

对于 XML 文件的写入,Qt 也提供了三种可用的方法:

- 使用 QXmlStreamWriter。
- 在内存中以 DOM 树的结构表示数据,并要求这个树型结构将自己写到文件中。
- 手动生成 XML。

使用 QXmlStreamWriter 是目前最简单易行的方式,同时它也比手动生成 XML 文档更加可靠。使用 DOM 生成 XML 的方法,在 DOM 树已作为应用程序的基本数据结构时才真正有意义。本章将详细介绍这三种读写 XML 的方法。

16.1　使用 QXmlStreamReader 读取 XML

使用 QXmlStreamReader 是在 Qt 中读取 XML 文档的最快且最简单的方式。因为解析器的工作能力是逐渐递增的,所以它尤其适用于诸如查找 XML 文档中一个给定的标记符出现的次数、读取内存容纳不了的特大文件、组装定制的数据结构以反映 XML 文档的内容等。

QXmlStreamReader 解析器根据图 16.1 中所列出的记号(token)工作。每次只要调用 readNext() 函数,下一个记号就会被读取并变成当前的记号。当前记号的属性取决于记号的类型,可以使用表格中列出的 getter 函数读取当前记号。

① Qt 4.4 有望包含另外的一些高级类来处理 XML,以在单独的名为 QtXmlPatterns 的模块中为 XQuery 和 XPath 提供支持。

记号类型	示例	getter函数
StartDocument	N/A	isStandaloneDocument()
EndDocument	N/A	isStandaloneDocument()
StartElement	<item>	namespaceUri(), name(), attributes(), namespaceDeclarations()
EndElement	</item>	namespaceUri(), name()
Characters	AT&T	text(), isWhitespace(), isCDATA()
Comment	<!-- fix -->	text()
DTD	<!DOCTYPE ...>	text(), notationDeclarations(), entityDeclarations()
EntityReference	™	name(), text()
ProcessingInstruction	<?alert?>	processingInstructionTarget(), processingInstructionData()
Invalid	>&<!	error(), errorString()

图 16.1 QxmlStreamReader 的记号

考虑如下的 XML 文档:

```
<doc>
    <quote>Einmal ist keinmal</quote>
</doc>
```

如果解析这个文档,则 readNext()每调用一次都将生成一个新记号,若使用 getter 函数还会获得额外的信息:

```
StartDocument
StartElement (name() == "doc")
StartElement (name() == "quote")
Characters (text() == "Einmal ist keinmal")
EndElement (name() == "quote")
EndElement (name() == "doc")
EndDocument
```

每次调用 readNext()后,都可以使用 isStartElement()、isCharacters()及类似的函数或者仅仅用 state()来测试当前记号的类型。

下面将查看一个实例,它告诉我们如何使用 QXmlStreamReader 解析一个专门的 XML 文件格式并在 QTreeWidget 中显示其内容。所解析的是那种具有书刊索引目录且包含索引条目和子条目的文档格式。图 16.2 是在 QTreeWidget 中显示的书刊索引文件。

```
<?xml version="1.0"?>
<bookindex>
    <entry term="sidebearings">
        <page>10</page>
        <page>34-35</page>
        <page>307-308</page>
    </entry>
    <entry term="subtraction">
        <entry term="of pictures">
            <page>115</page>
            <page>244</page>
        </entry>
        <entry term="of vectors">
            <page>9</page>
        </entry>
    </entry>
</bookindex>
```

首先查看从应用程序的 main()函数中提取出的代码,以从中了解 XML 阅读器在上下文中是如何使用的。然后,我们将查看阅读器的实现代码。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringList args = QApplication::arguments();
    ...
    QTreeWidget treeWidget;
    ...
    XmlStreamReader reader(&treeWidget);
    for (int i = 1; i < args.count(); ++i)
        reader.readFile(args[i]);
    return app.exec();
}
```

在图 16.2 中显示的应用程序首先创建一个 QTreeWidget。之后，这个应用程序创建一个 XmlStreamReader，并将树形窗口部件值传递给该 XmlStreamReader 并要求它解析在命令行中所指定的每一个文件。

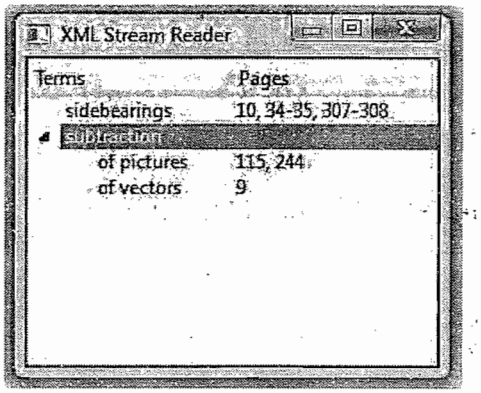


图 16.2　XML 流阅读器应用程序

```
class XmlStreamReader
{
public:
    XmlStreamReader(QTreeWidget *tree);

    bool readFile(const QString &fileName);

private:
    void readBookindexElement();
    void readEntryElement(QTreeWidgetItem *parent);
    void readPageElement(QTreeWidgetItem *parent);
    void skipUnknownElement();

    QTreeWidget *treeWidget;
    QXmlStreamReader reader;
};
```

XmlStreamReader 类提供了两个公共函数：构造函数和 parseFile()函数。这个类使用 QXmlStreamReader 实例解析 XML 文件，并组装 QTreeWidget 窗口以反映其读入的 XML 数据。通过使用向下递归的方法来实现这一解析过程。

- readBookindexElement()解析一个含有 0 或 0 个以上 <entry> 元素的 <bookindex> ... </bookindex> 元素。
- readEntryElement()解析一个含有 0 或 0 个以上 <page> 元素的 <entry> ... </entry> 元素，以及嵌套任意层次的含有 0 或 0 个以上 <entry> 元素。
- readPageElement()解析一个 <page> ... </page> 元素。
- skipUnknownElement()跳过不能识别的元素。

现在看看 XmlStreanReader 类的实现,由构造函数开始。

```
XmlStreamReader::XmlStreamReader(QTreeWidget *tree)
{
    treeWidget = tree;
}
```

构造函数只用来建立阅读器将使用的那个 QTreeWidget。所有的操作都将在 readFile()函数中完成(由 main()函数调用),我们将分三个部分来查看它们。

```
bool XmlStreamReader::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        std::cerr << "Error: Cannot read file " << qPrintable(fileName)
                  << ": " << qPrintable(file.errorString())
                  << std::endl;
        return false;
    }

    reader.setDevice(&file);
```

readFile()函数首先会尝试打开文件。如果失败,则会输出一条出错信息并返回 false 值;如果成功,则它将被设置为 QXmlStreamReader 的输入设备。

```
reader.readNext();
while (!reader.atEnd()) {
    if (reader.isStartElement()) {
        if (reader.name() == "bookindex") {
            readBookindexElement();
        } else {
            reader.raiseError(QObject::tr("Not a bookindex file"));
        }
    } else {
        reader.readNext();
    }
}
```

QXmlStreamReader 的 readNext()函数从输入流中读取下一个记号。如果成功而且还没有到达 XML 文件的结尾,函数将进入 while 循环。由于索引文件的结构,我们知道在该循环内部只有三种可能性发生:<bookindex> 开始标签正好被读入;另一个开始标签正好被读入(在这种情况下,读取的文件不是一个书刊索引);读入的是其他种类的记号。

如果有正确的开始标签,就调用 readBookindexElement()继续完成处理。否则,就调用 QXmlStreamReader::raiseError()并给出出错信息。下一次(在 while 循环条件下)调用 atEnd()时,它将返还 true 值。这就确保了解析过程可以在遇到错误时能尽快停止。通过对 QFile 调用 error()和 errorString(),就可以在稍后查询这些出错信息。当在书刊索引文件中检测到有错误时,也会立即返回一个类似的出错信息。其实,使用 raiseError()通常会更加方便,因为它对低级的 XML 解析错误和与应用程序相关的错误使用了相同的错误报告机制,而这些低级的 XML 解析错误会在 QXmlStreamReader 运行到无效的 XML 时就自动出现。

```
    file.close();
    if (reader.hasError()) {
        std::cerr << "Error: Failed to parse file "
                  << qPrintable(fileName) << ": "
                  << qPrintable(reader.errorString()) << std::endl;
        return false;
    } else if (file.error() != QFile::NoError) {
        std::cerr << "Error: Cannot read file " << qPrintable(fileName)
                  << ": " << qPrintable(file.errorString())
                  << std::endl;
        return false;
```

```
    }
    return true;
}
```

一旦处理完成,就会关闭文件。如果存在解析器错误或者文件错误,该函数就输出一个出错信息并返回 false 值;否则,返回 true 值并报告解析成功。

```
void XmlStreamReader::readBookindexElement()
{
    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }

        if (reader.isStartElement()) {
            if (reader.name() == "entry") {
                readEntryElement(treeWidget->invisibleRootItem());
            } else {
                skipUnknownElement();
            }
        } else {
            reader.readNext();
        }
    }
}
```

readBookindexElement()的作用就是读取文件的主体部分。它首先跳过当前的记号(此处只可能是 <bookindex> 开始标签),然后遍历读取整个输入文件。

如果读取到了关闭标签,那么它只可能是 </bookindex> 标签,否则 QXmlStreamReader 早就已经报告出错了(UnexpectedElementError)。如果是那样的话,就跳过这个标签并跳出循环。否则,将应该有一个顶级索引 <entry> 开始标签。如果情况确实如此,调用 readEntryElement()来处理条目数据;不然,就调用 skipUnknownElement()。使用 skipUnknownElement()而不调用 raiseError(),意味着如果要在将来扩展书刊索引格式以包含新的标签的话,这个阅读器将继续有效,因为它仅忽略了不能识别的标签。

readEntryElement()具有一个确认父对象条目的 QTreeWidgetItem * 参数。我们将 QTreeWidget::invisibleRootItem()作为父对象项传递,以使新的项以其为根基。在 readEntryElement()中,用一个不同的父对象项循环调用 readEntryElement()。

```
void XmlStreamReader::readEntryElement(QTreeWidgetItem *parent)
{
    QTreeWidgetItem *item = new QTreeWidgetItem(parent);
    item->setText(0, reader.attributes().value("term").toString());

    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }

        if (reader.isStartElement()) {
            if (reader.name() == "entry") {
                readEntryElement(item);
            } else if (reader.name() == "page") {
                readPageElement(item);
            } else {
                skipUnknownElement();
            }
        } else {
            reader.readNext();
```

```
        }
    }
}
```

每当遇到一个 <entry> 开始标签时,就会调用 readEntryElement()函数。我们希望为每一个索引条目创建一个树形的窗口部件项,因此创建一个新的 QTreeWidgetItem,并将其第一列的文本值设置为条目的项属性文本。

一旦条目被添加到树中,就开始读取下一个记号。如果这是一个关闭标签,就跳过该标签并跳出循环。如果遇到的是开始标签,那么它可能是 <entry> 标签(表示一个子条目), <page> 标签(该条目项的页码数),或者是一个未知的标签。如果开始标签是一个子条目,就递归调用 readEntryElement()。如果该标签是 <page> 标签,就调用 readPageElement()。

```
void XmlStreamReader::readPageElement(QTreeWidgetItem *parent)
{
    QString page = reader.readElementText();
    if (reader.isEndElement())
        reader.readNext();

    QString allPages = parent->text(1);
    if (!allPages.isEmpty())
        allPages += ", ";
    allPages += page;
    parent->setText(1, allPages);
}
```

只要读取的是 <page> 标签,就调用 readPageElement()函数。被传递的正是符合页码文本所属条目的树项。我们从读取 <page> 和 </page> 标签之间的文本开始。成功读取完以后,readElementText()函数将让解析器停留在必须跳过的 </page> 标签上。

这些页被存储在树形窗口部件项的第二列。我们首先提取那里已有的文本。如果文本不为空值,就在其后添加一个逗号,为新页的文本做好准备。然后,添加新的文本并相应地更新该列的文本。

```
void XmlStreamReader::skipUnknownElement()
{
    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }

        if (reader.isStartElement()) {
            skipUnknownElement();
        } else {
            reader.readNext();
        }
    }
}
```

最后,当遇到未知的标签时,将继续读取,直到读取到也将跳过的未知元素的关闭标签为止。这意味着我们将跳过那些具有良好形式但却无法识别的元素,并从 XML 文件中读取尽可能多的可识别的数据。

这里给出的实例可以作为类似的 XML 向下递归解析器的基础。然而,有时候实现这样一个解析器可能是相当棘手的,如果没有调用 readNext()或者在不恰当地方调用。一些程序员通过在代码中使用断言(assertion)来强调这个问题。例如,在 readBookindexElement()的开头,我们可以加上一行代码:

```
Q_ASSERT(reader.isStartElement() && reader.name() == "bookindex");
```

也可以在 readEntryElement()和 readPageElement()函数中使用类似的断言。对于 skipUnknownElement(),我们将仅声明存在一个开始元素。

QXmlStreamReader 可以从包括 QFile、QBuffer、QProcess 和 QTcpSocket 的任意 QIODevice 中获得输入。一些输入数据源可能无法在解析器需要的时候提供其所需要的数据,例如由于网络等待时间所造成。但在这种情况下仍可使用 QXmlStreamReader。在"Incremental Parsing"主题下关于 QXmlStreamReader 的参考文档中,提供了关于这方面的更多信息。

在这个应用程序中使用的 QXmlStreamReader 类是 QtXml 库中的一部分。如果想要建立应用程序和 QtXml 库的关联,必须在 .pro 文件中加入如下一行命令:

```
QT += xml
```

在接下来的两节中,将看到如何使用 DOM 和 SAX 来编写相同的应用程序。

16.2　用 DOM 读取 XML

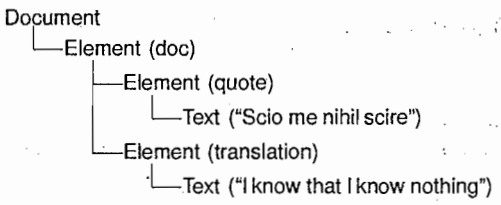
DOM 是一种解析由万维网协会(W3C)所开发的 XML 文档的标准应用程序编程接口。Qt 提供一套用于读取、操作和编写 XML 文档的非验证型二级 DOM 实现。

DOM 把 XML 文件表示成内存中的一棵树。我们可以按需要遍历这个 DOM 树,也可以修改这个树并把它作为 XML 文件保存到磁盘中。

让我们考虑如下这个 XML 文档:

```
<doc>
    <quote>Scio me nihil scire</quote>
    <translation>I know that I know nothing</translation>
</doc>
```

它对应如下所示的 DOM 树:



这个 DOM 树包含不同类型的节点。例如,Element 节点对应打开标签以及与它匹配的关闭标签。在这两个标签之间的内容则作为这个 Element 节点的子节点出现。在 Qt 中,节点类型(和其他所有与 DOM 有关的类一样)具有一个 QDom 前缀。因此,QDomElement 就代表一个 Element 节点,而 QDomText 就代表一个 Text 节点。

不同类型的节点可以具有不同种类的子节点。例如,一个 Element 节点可以包含其他 Element 节点,也可以包含 EntityReference、Text、CDATASection、ProcessingInstruction 以及 Comment 节点。图 16.3 给出了节点可以包含在的子节点的种类。图中显示为灰色的节点则不能拥有它自己的子节点。

为了演示如何使用 DOM 读取 XML 文件,我们将为前一节(见 292 页)中提到的书刊索引文件格式编写一个解析器。

```
class DomParser
{
public:
    DomParser(QTreeWidget *tree);
```

```
    bool readFile(const QString &fileName);

private:
    void parseBookindexElement(const QDomElement &element);
    void parseEntryElement(const QDomElement &element,
                           QTreeWidgetItem *parent);
    void parsePageElement(const QDomElement &element,
                          QTreeWidgetItem *parent);

    QTreeWidget *treeWidget;
};
```

定义了一个称为 DomParser 的类,它将会解析 XML 书刊索引文档并且在 QTreeWidget 中显示结果。

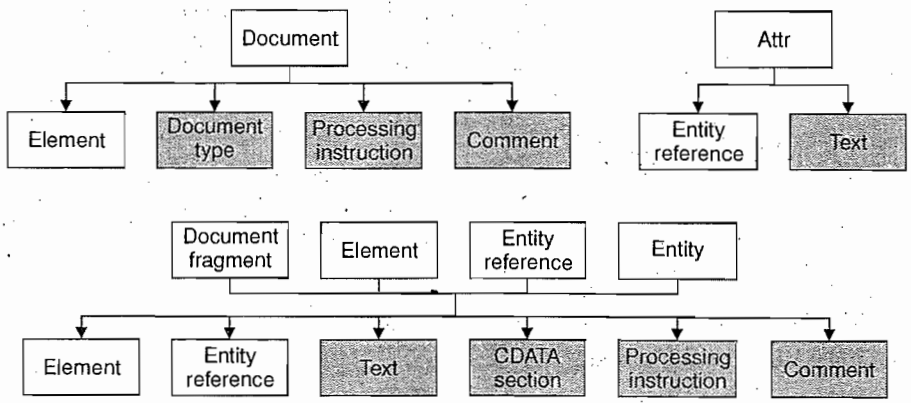


图 16.3 DOM 节点间的父子关系

```
DomParser::DomParser(QTreeWidget *tree)
{
    treeWidget = tree;
}
```

在构造函数中,我们仅将给定的树形窗口部件赋给成员变量。所有的解析都在 readFile()函数的内部完成。

```
bool DomParser::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        std::cerr << "Error: Cannot read file " << qPrintable(fileName)
                  << ": " << qPrintable(file.errorString())
                  << std::endl;
        return false;
    }

    QString errorStr;
    int errorLine;
    int errorColumn;

    QDomDocument doc;
    if (!doc.setContent(&file, false, &errorStr, &errorLine,
                        &errorColumn)) {
        std::cerr << "Error: Parse error at line " << errorLine << ", "
                  << "column " << errorColumn << ": "
                  << qPrintable(errorStr) << std::endl;
        return false;
    }

    QDomElement root = doc.documentElement();
    if (root.tagName() != "bookindex") {
```

```
        std::cerr << "Error: Not a bookindex file" << std::endl;
        return false;
    }

    parseBookindexElement(root);
    return true;
}
```

在 readFile()中,首先尝试打开那些文件名已经被传递进来的文件。如果有错误发生,就输出一个出错信息,并返回 false 值表示文件打开失败。否则,就设置一些变量用以保存那些需要的解析出错信息,然后创建一个 QDomDocument。当对 DOM 文档调用 setContent()函数时,由 QIODevice 提供的整个 XML 文档将被读取并解析。如果该文档还未打开,setContent()函数将自动打开设备。setContent()的 false 参数将禁用命名空间的处理。关于 XML 命名空间及其如何在 Qt 中处理,可以查阅 QtXml 的参考文档。

如果有错误发生,就输出一个出错信息并返回 false 值表示解析失败。如果解析成功,就对 QDomDocument 调用 documentElement()以获得它唯一的 QDomElement 子对象,同时检查它是否为 <bookindex> 元素。如果已经有 <bookindex> 元素了,就调用 parseBookindexElement()来解析它。与前一节中介绍的内容相似,解析过程是使用向下递归方法来实现的。

```
void DomParser::parseBookindexElement(const QDomElement &element)
{
    QDomNode child = element.firstChild();
    while (!child.isNull()) {
        if (child.toElement().tagName() == "entry")
            parseEntryElement(child.toElement(),
                              treeWidget->invisibleRootItem());
        child = child.nextSibling();
    }
}
```

在 parseBookindexElement()中,遍历所有的子节点。我们希望每一个节点都是一个 <entry> 元素,那样的话,就可以调用 parseEntry()来解析每一个节点。我们忽略那些未知的节点,以使书刊索引格式在不阻止旧的解析器工作的情况下今后也能被扩展。所有的 <entry> 节点都是 <bookindex> 节点的直接子对象,在组装的用于反映 DOM 树的窗口部件中它还是顶级节点。所以当我们想逐一解析这些节点时,将传递节点元素和树的不可见根项以作为窗口部件树项的父对象。

QDomNode 类可以存储任何类型的节点。如果想进一步处理一个节点,首先必须把它转换为正确的数据类型。在这个实例中,我们仅仅关心 Element 节点,所以对 QDomNode 调用 toElement()以把它转换成 QDomElement,然后调用 tagName()来取得元素的标签名称。如果节点不是 Element 类型,那么 toElement()函数就返回一个空 QDomElement 对象和一个空的标签。

```
void DomParser::parseEntryElement(const QDomElement &element,
                                  QTreeWidgetItem *parent)
{
    QTreeWidgetItem *item = new QTreeWidgetItem(parent);
    item->setText(0, element.attribute("term"));

    QDomNode child = element.firstChild();
    while (!child.isNull()) {
        if (child.toElement().tagName() == "entry") {
            parseEntryElement(child.toElement(), item);
        } else if (child.toElement().tagName() == "page") {
            parsePageElement(child.toElement(), item);
        }
        child = child.nextSibling();
    }
}
```

在 parseEntryElement()中,我们创建一个树形窗口部件项。传入的父对象项既可以是树的不可见根项(如果这是一个顶级条目的话),也可以是其他的条目(如果它只是一个子条目的话)。我们调用 setText(),将显示在项的第一列中的文本设置为 <entry> 标签的 term 属性值。

一旦已经初始化了 QTreeWidgetItem,就遍历对应于当前 <entry> 标签的 QDomElement 节点下的所有子节点。对于每一个 <entry> 标签的子元素,我们利用当前项作为第二个参数来递归调用 parseEntryElement()。然后,利用当前条目作为父对象,创建每一子节点的 QTreeWidgetItem。如果子元素为 <page>,就调用 parsePageElement()。

```
void DomParser::parsePageElement(const QDomElement &element,
                                 QTreeWidgetItem *parent)
{
    QString page = element.text();
    QString allPages = parent->text(1);
    if (!allPages.isEmpty())
        allPages += ", ";
    allPages += page;
    parent->setText(1, allPages);
}
```

在 parsePageElement()中,对元素调用 text()以获得 <page> 和 </page> 标签之间文本;然后,将文本添加到 QTreeWidgetItem 的第二列,这是一列以逗号分隔的页码列表。QDomElement::text()函数遍历元素的所有子节点并连接存储在 Text 和 CDATA 节点中的所有文本。

现在看看如何使用 DomParser 类来解析文件:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringList args = QApplication::arguments();
    ...
    QTreeWidget treeWidget;
    ...
    DomParser parser(&treeWidget);
    for (int i = 1; i < args.count(); ++i)
        parser.readFile(args[i]);

    return app.exec();
}
```

首先设立一个 QTreeWidget,然后再创建一个 DomParser。对于命令行中列出的每一个文件,都调用 DomParser::readFile()来打开并解析它,同时组装树形窗口部件。

与前面的例子相似,为了连接到 QtXml 库,需要将如下命令行添加到应用程序的 .pro 文件中:

```
QT += xml
```

就像这个实例所描绘的,尽管并不如使用 QXmlStreamReader 那么方便,遍历一个 DOM 树也还是相当简单和直接的。频繁使用 DOM 的程序员会经常编写他们自己的高级封装函数来简化那些常规需求的操作。

16.3 使用 SAX 读取 XML

SAX 事实上是公共领域中一种用于读取 XML 文档的标准应用程序编程接口。Qt 的 SAX 类是对基于 SAX2 的 Java 实现的模拟,只是在命名上有些不太符合 Qt 的惯例。与 DOM 相比,SAX 更加底层但通常也更加快速。然而,由于在本章前面部分曾介绍过的 QXmlSimpleReader 类提供了一个更接近 Qt 风格的应用程序编程接口,且比 SAX 解析器更加快速,因此 SAX 解析器的主要用途就是

将使用 SAX 应用程序编程接口的代码导入 Qt 中。有关 SAX 更详细的信息，请参看 http://www.sax-project.org/。

Qt 提供了一个名为 QXmlSimpleReader 的基于 SAX 的非验证型 XML 解析器。这个解析器能够识别具有良好格式的 XML 文档并且支持 XML 文档的命名空间。当这个解析器遍历文档时，它调用注册的处理函数中的虚拟函数来表明解析事件。(这些“解析事件”和 Qt 事件并无联系，就像按键事件和鼠标事件一样)。我们假设这个解析器正在解析如下的 XML 文档：

```
<doc>
    <quote>Gnothi seauton</quote>
</doc>
```

解析器将会调用如下这些解析事件处理函数：

```
startDocument()
startElement("doc")
startElement("quote")
characters("Gnothi seauton")
endElement("quote")
endElement("doc")
endDocument()
```

上述的这些函数都是在 QXmlContentHandler 中声明过的。为了简单，我们省略了 startElement() 和 endElement()中的一些参数。

QXmlContentHandle 只是可以和 QXmlSimpleReader 协作使用的众多处理程序类之一。其他的还有 QXmlEntityResolver、QXmlDTDHandler、QXmlErrorHandler、QXmlDeclHandler 和 QXmlLexicalHandler。这些类仅仅声明纯虚函数并且给出不同类型的解析事件的相关信息。对于绝大多数应用程序来说，只有 QXmlContentHandler 和 QXmlErrorHandler 是必要的。我们用到的类的层级关系如图 16.4 所示。

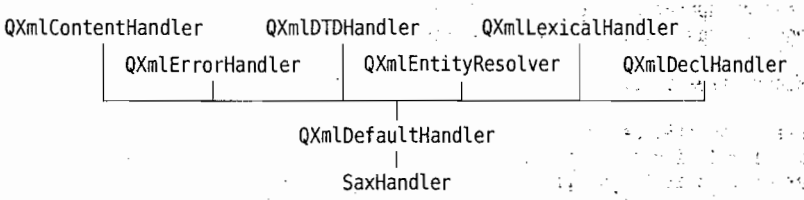


图 16.4　SaxHandler 的继承树

为了方便，Qt 还提供了 QXmlDefaultHandler，这是一个派生自所有处理程序类并且为所有函数都提供具体实现的类。含有很多抽象处理程序类和一个具体子类的这种设计构思，在 Qt 中并不常用，这里它被用来密切关注模型的 Java 实现。

与使用 QXmlStreamReader 或者 DOM 应用程序编程接口相比，使用 SAX 应用程序编程接口最显著的区别在于：SAX 应用程序编程接口需要我们利用成员变量手动追踪解析器的状态，而其他两种采用考虑向下递归的方法则不需要。

为了阐明如何使用 SAX 读取 XML 文件，我们将为本章前面部分提到的书刊索引文件格式编写一个解析器。这里将使用一个 QXmlSimpleReader 和一个名为 SaxHandler 的 QXmlDefaultHandler 子类来解析。

实现解析器的第一步是定义子类 QXmlDefaultHandler：

```
class SaxHandler : public QXmlDefaultHandler
{
public:
    SaxHandler(QTreeWidget *tree);

    bool readFile(const QString &fileName);
```

```
protected:
    bool startElement(const QString &namespaceURI,
                      const QString &localName,
                      const QString &qName,
                      const QXmlAttributes &attributes);
    bool endElement(const QString &namespaceURI,
                    const QString &localName,
                    const QString &qName);
    bool characters(const QString &str);
    bool fatalError(const QXmlParseException &exception);

private:
    QTreeWidget *treeWidget;
    QTreeWidgetItem *currentItem;
    QString currentText;
};
```

SaxHandler 类派生自 QXmlDefaultHandler,并且重新实现了 4 个函数:startElement()、endElement()、characters()和 fatalError()。前面三个函数都是在 QXmlContentHandler 中声明的,最后一个函数则是在 QXmlErrorHandler 中声明的。

```
SaxHandler::SaxHandler(QTreeWidget *tree)
{
    treeWidget = tree;
}
```

SaxHandler 构造函数接受用存储在 XML 文件中的信息进行组装的 QTreeWidget。

```
bool SaxHandler::readFile(const QString &fileName)
{
    currentItem = 0;

    QFile file(fileName);
    QXmlInputSource inputSource(&file);
    QXmlSimpleReader reader;
    reader.setContentHandler(this);
    reader.setErrorHandler(this);
    return reader.parse(inputSource);
}
```

当获得要解析文件的文件名时,就会调用这个函数。我们为文件创建一个 QFile 对象,同时创建一个 QXmlInputSource 以读取文件的内容。然后,创建一个 QXmlSimpleReader 来解析这个文件。我们设置该类(SaxHandler)的阅读器内容和错误处理程序,接着对阅读器调用 parse()来执行解析。在 SaxHandler 中,仅重新实现了来自于 QXmlContentHandler 和 QXmlErrorHandler 类的函数。如果已经重新实现了来自于其他处理程序类的函数,则还需要调用相应的 set*Xxx*Handler()函数。

我们传递一个 QXmlInputSource,而不是传递一个简单的 QFile 对象给 parse()函数。这个类将打开并读取给定的文件(考虑了 <?xml? > 声明中指定的任意字符编码)给定的文件,同时它还提供了一个解析器读取文件的接口。

```
bool SaxHandler::startElement(const QString & /* namespaceURI */,
                              const QString & /* localName */,
                              const QString &qName,
                              const QXmlAttributes &attributes)
{
    if (qName == "entry") {
        currentItem = new QTreeWidgetItem(currentItem ?
                currentItem : treeWidget->invisibleRootItem());
        currentItem->setText(0, attributes.value("term"));
    } else if (qName == "page") {
        currentText.clear();
    }
    return true;
}
```

当阅读器遇到一个新的打开标签时，就会调用 startElement()函数。第三个参数是标签的名称(或者更加准确地说，是它的“限定名”)。第四个参数是属性列表。在这个实例中，忽略了第一个和第二个参数。对于使用 XML 命名空间机制的 XML 文件，它们非常有用，这个主题会在参考文档中详细论述。

如果标签是 <entry>，就创建一个新的 QTreeWidgetItem 项。如果标签嵌套在另一个 <entry> 中，则新的标签将在这个索引中定义一个子条目，并且这个新的 QTreeWidgetItem 会作为代表包含条目的 QTreeWidgetItem 的子对象而创建。不然，就创建 QTreeWidgetItem 作为顶级项，使用树形窗口部件的不可见根项作为它的父对象。我们调用 setText()，将第 0 列中显示的文本值设置为这个 <entry> 标签的 term 属性。

如果标签是 <page>，就将 currentText 变量设置为一个空字符串。该变量作为一个累加器，用于 <page> 和 </page> 标签之间的文本。

最后，我们返回 true 值让 SAX 继续解析这个文件。如果想把那些未知的标签也作为错误报告，这时就需要返回 false 值。然后，还可以在 QXmlDefaultHandler 中重新实现 errorString()，以返回一个适当的出错消息。

```
bool SaxHandler::characters(const QString &str)
{
    currentText += str;
    return true;
}
```

可以调用函数 characters()报告 XML 文档中的字符数据。我们只把这些字符添加到 currentText 变量中。

```
bool SaxHandler::endElement(const QString & /* namespaceURI */,
                            const QString & /* localName */,
                            const QString &qName)
{
    if (qName == "entry") {
        currentItem = currentItem->parent();
    } else if (qName == "page") {
        if (currentItem) {
            QString allPages = currentItem->text(1);
            if (!allPages.isEmpty())
                allPages += ", ";
            allPages += currentText;
            currentItem->setText(1, allPages);
        }
    }
    return true;
}
```

当阅读器遇到一个关闭标签时，就会调用 endElement()函数。就像 startElement()一样，第三个参数是标签的名称。

如果标签是 </entry>，就更新 currentItem 私有变量，使它指向当前 QTreeWidgetItem 的父对象。(因为一些历史原因，顶级项返回 0 值作为它们的父对象，而不是返回不可见的 root 项。)这样可以确保 currentItem 变量能恢复为相应的 <entry> 标签被读取之前所保存的值。

如果标签为 </page>，则把指定的页码或者页码范围添加到第一列当前项的文本中以逗号分隔的列表中。

```
bool SaxHandler::fatalError(const QXmlParseException &exception)
{
    std::cerr << "Parse error at line " << exception.lineNumber()
              << ", " << "column " << exception.columnNumber() << ": "
              << qPrintable(exception.message()) << std::endl;
    return false;
}
```

当阅读器解析 XML 文件失败时,就会调用 fatalError()函数。如果这种情况发生,我们仅向控制台输出一条出错信息,给出行号、列号以及这个解析器的错误文本。

这样就完成了 SaxHandler 类的实现。它的 main()函数与前一节中解析 DomParser 的主函数几乎一样,唯一的区别在于这里使用的是 SaxHandler 而不是 DomParser。

16.4 写入 XML

能读取 XML 文件的大多数应用程序也需要写入 XML 文件。一般来说,主要有三种由 Qt 应用程序生成 XML 文件的方法:

- 使用 QXmlStreamWriter。
- 构建 DOM 树并对它调用 save()。
- 手动生成 XML。

实际上,这三种方法的选取通常与是否使用 QXmlStreamWriter、DOM 或 SAX 来读取 XML 文档并无关系,尽管数据保留在 DOM 树中时直接保存这个 DOM 树是完全说得通的。

利用 QXmlStreamWrite 类来写入 XML 文件非常容易,因为它将时刻为我们关注那些特殊的转义字符。如果想利用 QXmlStreamWrite 从 QTreeWidget 中输出书刊索引数据,只需要使用两个函数。第一个函数获得文件名和一个 QTreeWidget *,并且将遍历树中所有的顶级项:

```
bool writeXml(const QString &fileName, QTreeWidget *treeWidget)
{
    QFile file(fileName);
    if (!file.open(QFile::WriteOnly | QFile::Text)) {
        std::cerr << "Error: Cannot write file "
                  << qPrintable(fileName) << ": "
                  << qPrintable(file.errorString()) << std::endl;
        return false;
    }

    QXmlStreamWriter xmlWriter(&file);
    xmlWriter.setAutoFormatting(true);
    xmlWriter.writeStartDocument();
    xmlWriter.writeStartElement("bookindex");
    for (int i = 0; i < treeWidget->topLevelItemCount(); ++i)
        writeIndexEntry(&xmlWriter, treeWidget->topLevelItem(i));
    xmlWriter.writeEndDocument();
    file.close();
    if (file.error()) {
        std::cerr << "Error: Cannot write file "
                  << qPrintable(fileName) << ": "
                  << qPrintable(file.errorString()) << std::endl;
        return false;
    }
    return true;
}
```

如果开启了自动格式编辑功能,XML 文档将以更友好易读的格式输出,在每一行都有明确显示数据递归结构的缩进。writeStartDocument()函数则写下 XML 文档首行:

```
<?xml version="1.0" encoding="UTF-8"?>
```

writeStartElement()函数随给定的标签文本生成一个新的开始标签。writeStartDocument()函数则关闭任何打开的开始标签。对于每一个顶级项,我们调用 writeIndexEntry()函数,并将 QXmlStreamWriter 和要输出的项传递给 writeIndexEntry()函数。下面是 writeIndexEntry()的代码:

```
void writeIndexEntry(QXmlStreamWriter *xmlWriter, QTreeWidgetItem *item)
{
    xmlWriter->writeStartElement("entry");
```

```
        xmlWriter->writeAttribute("term", item->text(0));
        QString pageString = item->text(1);
        if (!pageString.isEmpty()) {
            QStringList pages = pageString.split(", ");
            foreach (QString page, pages)
                xmlWriter->writeTextElement("page", page);
        }
        for (int i = 0; i < item->childCount(); ++i)
            writeIndexEntry(xmlWriter, item->child(i));
        xmlWriter->writeEndElement();
    }
```

writeIndexEntry()函数创建一个对应于 QTreeWidgetItem 的 <entry> 元素并将其作为参数接收。writeAttribute()函数则为刚刚写入的标签添加一个属性。例如,它可以将 <entry> 转变为 <entry term ="sidebearings"> 。如果其中有页码,那么页码将以逗号-空格分隔,并且对于每一个页码数,会有一个单独的 <page> ... </page> 标签对与其间的页码文本一起被写入。这些都可以通过调用 writeTextElement()以及将标签名以及开始和结束标签之间的文本传递给 writeTextElement()来实现。在所有的情况中,QXmlStreamWriter 都考虑了特殊的 XML 转义字符的处理,我们不必为此担心。

如果项中有子项,就对每一个子项递归调用 writeIndexEntry()。最后,我们调用 writeEndElement()来输出 </entry> 。

使用 QXmlStreamWirter 是写入 XML 文档最容易、最安全的方式,但如果已经在一个 DOM 树中有 XML 文档的话,则只需要在 QDomDocument 对象上调用 save()函数并要求这个 DOM 树输出相关的 XML 即可。默认情况下,save()使用 UTF-8 作为所生成文件的编码方式。我们可以通过为 DOM 树预先进行 <?xml?> 声明,而使用其他种类的编码方式,例如:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

下面的代码段给出了其完成过程:

```
const int Indent = 4;

QDomDocument doc;
...
QTextStream out(&file);
QDomNode xmlNode = doc.createProcessingInstruction("xml",
                                "version=\"1.0\" encoding=\"ISO-8859-1\"");
doc.insertBefore(xmlNode, doc.firstChild());
doc.save(out, Indent);
```

从 Qt 4.3 开始,一个替代的方式就是利用 setCodec()对 QTextStream 设置编码,并将 QDomNode::EncodingFromTextStream 作为第三个参数传递给 save()。

手动生成 XML 文件并不比利用 DOM 生成 XML 文件更加困难。我们可以使用 QTextStream 并且写入字符串,就像写入其他任何文本文件一样。最需要慎重对待的部分就是转义在文本和属性值中的特殊字符。Qt::escape()函数可以转义“<”、“>”和“&”等特殊字符。以下是一个使用它的代码:

```
QTextStream out(&file);
out.setCodec("UTF-8");
out << "<doc>\n"
    << "    <quote>" << Qt::escape(quoteText) << "</quote>\n"
    << "    <translation>" << Qt::escape(translationText)
    << "</translation>\n"
    << "</doc>\n";
```

当采用这样的方式生成 XML 文件时,除了必须写入正确的 <?xml?> 声明并设置正确的编码外,还必须记得对写入的文本进行转义。如果使用了属性值,还必须转义其中的单引号或双引号。不过,使用 QXmlStreamWriter 就会简单得多了,因为它将为我们处理上述所需注意的一切。

第 17 章　提供在线帮助

绝大多数应用程序都为用户提供在线帮助。有一些帮助的内容是相当简短的，例如工具提示、状态提示和“What's This?”之类的帮助。Qt 自然都支持这类简单的帮助。其他帮助可以具有更多的扩展内容，例如包括很多页的文本。对于这种帮助，可以把 QTextBrowser 作为一个简单的在线帮助浏览器来使用，也可以从应用程序调用 Qt Assistant 或者其他 HTML 浏览器。

17.1　工具提示、状态提示和“What's This?”帮助

工具提示就是一小段文本，当鼠标在一个窗口部件上面停留一段时间后会显示。工具提示通常显示为黄色背景黑色文字，它的主要用途是为工具条按钮提供文本描述。

可以使用 QWidget::setToolTip()以代码的形式为任意窗口部件添加工具提示。例如：

```
findButton->setToolTip(tr("Find next"));
```

为了给一个能添加到主菜单或者工具条的 QAction 设置工具提示，只需对这个动作调用 setToolTip()。例如：

```
newAction = new QAction(tr("&New"), this);
newAction->setToolTip(tr("New document"));
```

如果没有明确地设置工具提示，QAction 会自动使用这个动作的文本。

状态提示也是一小段描述性文本，通常会比工具提示长一点。当鼠标在工具条按钮或者菜单选项上停留一段时间时，状态提示将会出现在状态栏上。调用 setStatusTip()可以为动作或窗口部件添加状态提示：

```
newAction->setStatusTip(tr("Create a new document"));
```

图 17.1 给出了应用程序的工具提示和状态提示。

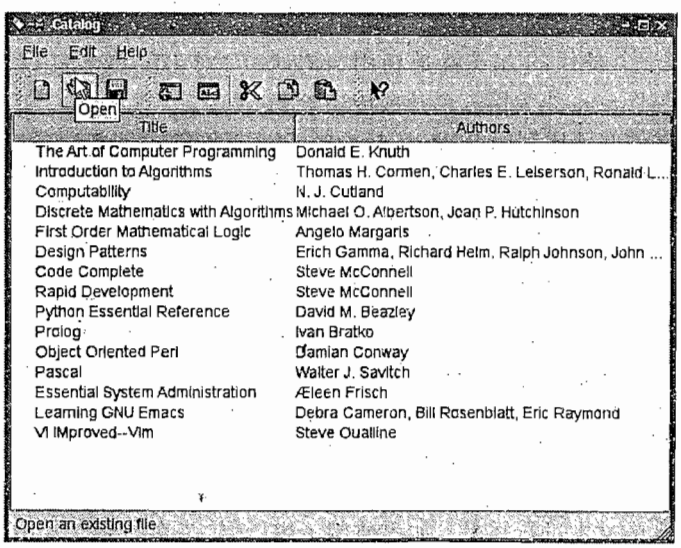


图 17.1　显示工具提示和状态提示的应用程序

在某些情况下，如果能为窗口部件提供比工具提示和状态提示更多的信息，则是最好不过的了。例如，我们也许想为一个复杂对话框中的每一个字段提供说明性的文本，而不强制用户调用一个单独的帮助窗口。“What's This?”模式是对这个问题的一个理想的解决方案。当窗口处于“What's This?”模式的时候，光标将会变为?的形式，而且用户可以在任何用户界面组件上单击来获得关于它的帮助文本。为了进入“What's This?”模式，用户既可以单击对话框标题栏(在 Windows 和 KDE 中)中的“?”按钮，也可以按下 Shift + F1 组合键。

以下是一个在对话框上设置“What's This?”文本帮助的例子：

```
dialog->setWhatsThis(tr("<img src=\":/images/icon.png\">"
                        " The meaning of the Source field depends "
                        "on the Type field:"
                        "<ul>"
                        "<li><b>Books</b> have a Publisher"
                        "<li><b>Articles</b> have a Journal name with "
                        "volume and issue number"
                        "<li><b>Theses</b> have an Institution name "
                        "and a Department name"
                        "</ul>"));
```

可以使用 HTML 标签来对“What's This?”文本进行格式化。在图 17.2 所示的实例中，我们包含了一个图像(它位于这个应用程序的资源文件中)、一个含有项目符号的列表以及一些加粗的文本。Qt 所支持的标签和属性在 http://doc.trolltech.com/4.3/richtext-html-subset.html 网页中有详细介绍。

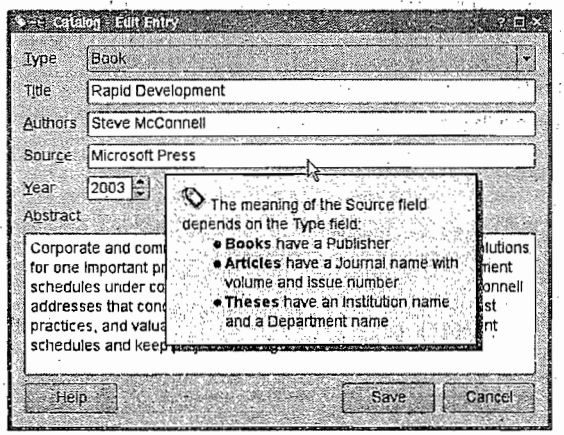


图 17.2 显示“What's This?”帮助文本的对话框

当在动作上设置了“What's This?”文本，且用户在“What's This?”模式下单击这个菜单项或者工具栏按钮或者按下快捷键的时候，这个文本将会显示出来。当应用程序的主窗口的用户界面组件提供“What's This?”文本帮助时，通常的惯例是在 Help 菜单中提供一个“What's This?”选项以及一个对应的工具栏按钮。通过利用静态函数 QWhatsThis::createAction()创建一个“What's This?”动作并将它返回的动作添加到 Help 菜单和工具栏，就可以完成以上的操作。QWhatsThis 类还提供了静态函数以有计划地进入和退出“What's This?”模式。

17.2 利用 Web 浏览器提供在线帮助

大型的应用程序需要提供比工具提示、状态提示和“What's This?”帮助更多的在线帮助信息。对于这个问题的一个简单的解决方案是提供 HTML 格式的帮助文本，并且在适当的页面中启动用户的 Web 浏览器。

包含帮助浏览器的应用程序通常在主窗口的 Help 菜单中有一个 Help 条目,并且在每一个相应的对话框中也有 Help 按钮。本节将介绍如何使用 QDesktopServices 类来为这些按钮提供相应的功能。

当用户按下 F1 键或单击 Help→Help 菜单选项时,应用程序的主窗口通常会调用一个 help()槽。

```
void MainWindow::help()
{
    QUrl url(directoryOf("doc").absoluteFilePath("index.html"));
    url.setScheme("file");
    QDesktopServices::openUrl(url);
}
```

在这个实例中,我们假设应用程序的 HTML 帮助文件都在一个名为 doc 的子目录中。QDir::absoluteFilePath() 函数返回一个具有给定文件名完全路径的 QString。首先创建一个具有帮助文件路径的 QUrl 对象。因为这是针对主窗口的帮助,所以使用自己的帮助系统的 index.html 文件,在这个文件中其他的帮助文件都可以通过超链接来访问。然后,我们设置 URL 的模式为“文件”,这样设置的文件就只会在本地文件系统中被查找。最后,使用桌面服务的 openUrl()静态方便函数来启动用户的 Web 浏览器。

由于我们并不知道将使用哪一个 Web 浏览器,所以必须小心确保我们的 HTML 有效并同时保持其与用户正使用的浏览器相兼容。大多数 Web 浏览器将把它们本地工作目录设置为 URL 路径,同时还假设任何没有绝对路径的图像和超链接都把工作目录作为其根目录。这意味着我们必须将所有的 HTML 文件和图像文件放在 doc 目录(或者其下的子目录中),同时关联所有索引,但那些连到外部站点的链接除外。

```
QDir MainWindow::directoryOf(const QString &subdir)
{
    QDir dir(QApplication::applicationDirPath());

#if defined(Q_OS_WIN)
    if (dir.dirName().toLower() == "debug"
            || dir.dirName().toLower() == "release")
        dir.cdUp();
#elif defined(Q_OS_MAC)
    if (dir.dirName() == "MacOS") {
        dir.cdUp();
        dir.cdUp();
        dir.cdUp();
    }
#endif
    dir.cd(subdir);
    return dir;
}
```

静态 directoryOf()函数返回一个对应于与应用程序目录相关的指定子目录的 QDir。在 Windows 中,应用程序的可执行文件通常位于 debug 或 release 子目录中,这样我们就往上移动一级目录。在 Mac OS X 中,我们主要考虑卷式(bundle)目录结构。

对于对话框,通常希望在帮助系统内部的某一具体页,或许也可能在某一页面中的某一具体点上启动 Web 浏览器。例如:

```
void EntryDialog::help()
{
    QUrl url(directoryOf("doc").absoluteFilePath("forms.html"));
    url.setScheme("file");
    url.setFragment("editing");
    QDesktopServices::openUrl(url);
}
```

当用户单击对话框的 Help 按钮时，这个槽是从对话框的内部调用的。除了已经选择了一个不同的开始页面以外，这与之前的实例非常相似。这个专门的页面有针对几个不同窗体的帮助文本，同时可以让 HTML 中的锚定引用(如)指定出每一个窗体的帮助文本从哪里开始。为了访问页面中某一具体的位置，我们调用 setFragment()并传递希望页面滚动至的锚定。

提供 HTML 格式的帮助文件并且让用户可以通过他们自己的 Web 浏览器去读取这些帮助文件，非常简单且方便。但是 Web 浏览器不能访问应用程序资源(比如图标等)，并且它们并不易于被用户定制以符合应用程序的要求。同样，如果我们像处理 EntryDialog 时那样跳到了某一页面，单击浏览器的 Home 或 Back 按钮则无法达到预期的效果。

17.3　将 QTextBrowser 作为简单的帮助引擎

通过使用用户的 Web 浏览器来显示在线帮助是非常容易的，但正如我们已经提到的，该方法也确实存在一些缺点。可以通过提供自己的基于 QTextBrowser 类的帮助引擎来消除这些问题。

这一节将给出一个如图 17.3 所示的简单的帮助浏览器，并且解释如何在应用程序中使用它。这个窗口使用 QTextBrowser 来显示标记着基于 HTML 语法的帮助页。QTextBrowser 可以处理很多简单的 HTML 标签，因此对于这个目的，它是相当令人满意的。

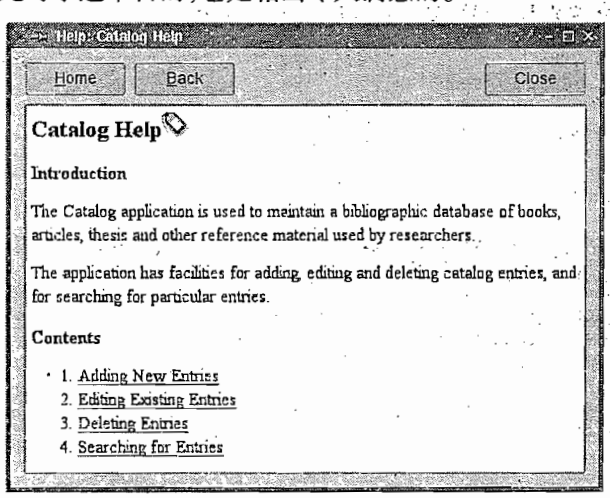


图 17.3　HelpBrowser 窗口部件

我们从类的定义开始：

```
class HelpBrowser : public QWidget
{
    Q_OBJECT

public:
    HelpBrowser(const QString &path, const QString &page,
                QWidget *parent = 0);

    static void showPage(const QString &page);

private slots:
    void updateWindowTitle();

private:
```

```
    QTextBrowser *textBrowser;
    QPushButton *homeButton;
    QPushButton *backButton;
    QPushButton *closeButton;
};
```

HelpBrowser 提供了一个静态函数,它可以在应用程序的任何地方被调用。这个函数创建一个 HelpBrowser 窗口并且显示给定的页。

下面是其构造函数:

```
HelpBrowser::HelpBrowser(const QString &path, const QString &page,
                         QWidget *parent)
    : QWidget(parent)
{
    setAttribute(Qt::WA_DeleteOnClose);
    setAttribute(Qt::WA_GroupLeader);

    textBrowser = new QTextBrowser;

    homeButton = new QPushButton(tr("&Home"));
    backButton = new QPushButton(tr("&Back"));
    closeButton = new QPushButton(tr("Close"));
    closeButton->setShortcut(tr("Esc"));

    QHBoxLayout *buttonLayout = new QHBoxLayout;
    buttonLayout->addWidget(homeButton);
    buttonLayout->addWidget(backButton);
    buttonLayout->addStretch();
    buttonLayout->addWidget(closeButton);

    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addLayout(buttonLayout);
    mainLayout->addWidget(textBrowser);
    setLayout(mainLayout);

    connect(homeButton, SIGNAL(clicked()), textBrowser, SLOT(home()));
    connect(backButton, SIGNAL(clicked()),
            textBrowser, SLOT(backward()));
    connect(closeButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(textBrowser, SIGNAL(sourceChanged(const QUrl &)),
            this, SLOT(updateWindowTitle()));

    textBrowser->setSearchPaths(QStringList() << path << ":/images");
    textBrowser->setSource(page);
}
```

我们设置 Qt::WA_GroupLeader 属性,因为除了要从主窗口中弹出 HelpBrowser 窗口,还希望从模式对话框中也弹出 HelpBrowser 窗口。通常情况下,模式对话框不允许用户和这个应用程序中的其他任何窗口进行交互。然而,在请求帮助之后,用户就显然被允许与模式对话框和帮助浏览器进行交互。设置 Qt::WA_GroupLeader 属性才可能实现这种交互。

我们提供了两种搜索路径,第一种是包含应用程序文档的文件系统中的一个路径,第二种是图像资源的存储位置。HTML 可以在文件系统中以常规方式包含图像资源的引用,也可以通过使用一个以":/"开始的路径包含图像资源的引用。page 参数是文档文件的名称,有一个可选的 HTML 锚。

```
void HelpBrowser::updateWindowTitle()
{
    setWindowTitle(tr("Help: %1").arg(textBrowser->documentTitle()));
}
```

只要这个源页面发生变化,就会调用 updateWindowTitle()槽。documentTitle()函数返回页的 <title> 标签中指定的文本。

```
void HelpBrowser::showPage(const QString &page)
{
    QString path = directoryOf("doc").absolutePath();
    HelpBrowser *browser = new HelpBrowser(path, page);
    browser->resize(500, 400);
    browser->show();
}
```

在 showPage()静态函数中，创建并显示了 HelpBrowser 窗口。当用户关闭它时，会被自动删除，因为在 HelpBrowser 构造函数中设置了 Qt::WA_DeleteOnClose 属性。对于这个实例，我们假设这些帮助文档都放在应用程序的 doc 目录下。所有传递给 showPage()函数的页都将取自于这个子目录。

现在已经准备好从应用程序中调用帮助浏览器了。在应用程序的主窗口中，可以创建一个 Help 动作并把它连接到一个 help()槽，其过程如下：

```
void MainWindow::help()
{
    HelpBrowser::showPage("index.html");
}
```

这里假设主帮助文件的名称为 index.html。对于对话框，可以建立 Help 按钮与 help()槽的连接，其过程大致如下：

```
void EntryDialog::help()
{
    HelpBrowser::showPage("forms.html#editing");
}
```

这里我们还顺便看了一个不同的帮助文件 forms.html，并且将 QTextBrowser 滚动到 editing 锚。

也可以使用 Qt 资源系统将帮助文件以及与它们相关的图像直接嵌入到可执行文件中。实现这个目的需要做的改变就是为每一个希望嵌入的文件添加条目到应用程序的 .qrc 文件中，并且使用资源路径(例如，:/doc/forms.html # editing)。

在这个实例中，我们同时使用了两种方法，既嵌入了图标(因为应用程序本身也使用它)，但是同时又保持了 HTML 文档在文件系统中。这一方法的优点在于帮助文件可以独立于应用程序而更新，而且可以确保其能够找到应用程序的图标。

17.4　使用 Qt Assistant 提供强大的在线帮助

Qt Assistant 是由 Trolltech 公司提供的一个在线帮助应用程序。它的主要优点是支持索引和全文检索，而且它可以为多个应用程序处理多个文档集。为了使用 Qt Assistant，必须在应用程序中加入必要的代码，并且必须让 Qt Assistan 意识到帮助文档的存在①。

Qt 应用程序和 Qt Assistant 之间通信是由 QAssistantClient 类来处理的，它位于一个单独的库中。为了建立这个库与应用程序的连接，必须在应用程序的 .pro 文件中添加如下的命令行：

```
CONFIG += assistant
```

现在查看一个使用 Qt Assistant 的新 HelpBrowser 类的代码：

```
class HelpBrowser
{
public:
    static void showPage(const QString &page);
```

① Qt 4.4 有望以不断更新的帮助系统为特色，这使我们的帮助文档更加容易被集成进去。

```
private:
    static QDir directoryOf(const QString &subdir);

    static QAssistantClient *assistant;
};
```

以下是一个新的 helpbrowser.cpp 文件:

```
QAssistantClient *HelpBrowser::assistant = 0;

void HelpBrowser::showPage(const QString &page)
{
    QString path = directoryOf("doc").absoluteFilePath(page);
    if (!assistant)
        assistant = new QAssistantClient("");
    assistant->showPage(path);
}
```

QAssistantClient 构造函数接收一个路径字符串作为它的第一个参数,它可以用于定位 Qt Assistant 可执行文件。通过传递一个空路径字符串,表示让 QAssistantClient 在 PATH 环境变量中查找可执行文件。QAssistantClient 有一个 showPage()函数,它可以接收带一个可选 HTML 锚点的页面名称。

下一步是为文档准备目录和索引。这是通过创建一个 Qt Assistant 配置文件并且写入一个提供有关文档信息的 .dcf 文件实现的。这些在 Qt Assistant 的在线帮助中都有说明,所以这里就不再赘述了。

除了使用 Web 浏览器、QTextBrowser 或 Qt Assistant,还有一种方法就是使用与平台相关的方法来提供在线帮助。对于 Windows 应用程序,创建 Windows HTML Help 文件并且利用 Microsoft Internet Explorer 访问它们,也许是种不错的选择。可以使用 Qt 的 QProcess 类或者 ActiveQt 框架来实现这一点。在 Mac OS X 系统上,Apple Help 也提供了与 Qt Assistant 相似的功能。

现在,本书的第二部分即将结束,而本书第三部分中的章节将包含更多的关于 Qt 高级特性和专门特征的内容。总的来说,在第三部分中给出的 C++ 和 Qt 代码并不会比第二部分所见到的更难。对于读者而言,那些出现在陌生领域中的某些概念和思想才更具挑战性。

第三部分　Qt 高级

第 18 章　国际化
第 19 章　自定义外观
第 20 章　三维图形
第 21 章　创建插件
第 22 章　应用程序脚本
第 23 章　平台相关特性
第 24 章　嵌入式编程

第18章 国 际 化

除了采用英语和许多欧洲语系中的拉丁字符之外，Qt 4 也为世界上其他的文字系统提供了广泛支持：

- Qt 在整个应用程序编程接口及其内部都使用 Unicode。无论用于用户接口的是何种语言，应用程序都可以为所有的用户提供类似的支持。
- Qt 的文本引擎可以处理所有主要的非拉丁文文字系统，其中包括阿拉伯文(Arabic)、中文(Chinese)、西里尔文(Cyrillic)、希伯来文(Hebrew)、日文(Japanese)、韩文(Korean)、泰文(Thai)和印度文(Indic)。
- Qt 的布局引擎可以为从右到左的文本布局提供支持，比如对阿拉伯文和希伯来文的支持。
- 一些特定的语言在输入文本时要求使用特殊的输入法。比如像 QLineEdit 和 QTextEdit 这样的一些编辑器窗口部件，都可以与安装在用户系统中的任意输入法和谐地工作在一起。

通常情况下，只允许用户使用他们自己的本地语言输入文本是不够的，最好是要让整个用户界面允许翻译才行。Qt 让这些变得非常简单：只需使用 tr()函数把用户所有可见的字符串都封装起来(就像前面章节中所做的那样)，并且使用 Qt 的支持工具为所需的语言准备相应的翻译文件即可。Qt 为翻译人员提供了一个称为 Qt Linguist 的图形用户界面工具。配上两个通常由应用程序开发人员运行的命令行语句——lupdate 和 lrelease，Qt Linguist 就可以为翻译提供完美支持了。

对于大多数的应用程序，程序会在一开始启动的时候根据用户的本地设置载入所需的翻译文件。但是在少数情况下，对用户来说，要求程序在运行的时候也能够切换语言，可能也是很有必要的。而利用 Qt，使这一切都变得完全可能起来，虽然这的确还需要再做一些其他工作。在翻译文本的长度比原有文本的长度长的时候，也要多亏了 Qt 的布局系统，使得不同的用户接口部分会自动进行调整，从而可以为翻译文本腾出空间。

18.1 使用 Unicode

Unicode 是一种支持世界上绝大多数文字系统的字符编码标准。Unicode 最初的思想是在存储字符时使用 16 位而不是 8 位，这样它就可以大约具有 65 000 个字符编码，而不是只有 256 个[①]。Unicode 在同样的编码位置以子集的形式包含了 ASCII 和 ISO 8859-1(Latin-1)。例如，字符“A”的 ASCII 编码、Latin-1 编码和 Unicode 编码都是 0x41，而字符“?”的 Latin-1 编码和 Unicode 编码都是 0xD1。

Qt 中的 QString 类用于将字符串存储为 Unicode。在 QString 中的每一个字符都是一个 16 位的 QChar，而不是一个 8 位的 char。以下是把一个字符串的第一个字符设置成“A”的两种方法：

```
str[0] = 'A';
str[0] = QChar(0x41);
```

① 新版的 Unicode 标准可分配的字符数超过 65 535。使用两个称为“代理编码对”(surrogate pairs)的 16 位值序列，就可以表示所有这些字符。

如果源文件的编码格式是 Latin-1，那么指定 Latin-1 字符就非常简单：

```
str[0] = 'Ñ';
```

如果源文件中还使用了其他的编码格式，那么使用数字赋值的方式会更好些：

```
str[0] = QChar(0xD1);
```

我们可以使用任何一个 Unicode 字符的数字值来指定它。例如，以下给出了如何指定大写的希腊字符西格玛（“Σ”）和欧元货币符号（“€”）的方法：

```
str[0] = QChar(0x03A3);
str[0] = QChar(0x20AC);
```

Unicode 所支持的所有字符的数值列表可以在 http://www.unicode.org/standard 中找到。如果你很少需要非 Latin-1 的 Unicode 字符，在线查找这些字符就已经足够了。但是，为了在 Qt 程序中输入 Unicode 字符串，Qt 还是提供了很多简便的方法，正如我们在这一节的稍后部分将要看到的那样。

Qt 4 的文本引擎在所有平台上都支持以下文字系统：阿拉伯文、中文、西里尔文、希腊文、希伯来文、日文、韩文、老挝文、拉丁文、泰文和越南文。它也支持所有的 Unicode 4.1 脚本而不需要任何特殊处理。此外，在使用 Fontconfig 的 X11 和新版 Qt 的 Windows 系统上，Qt 还支持以下这些文字系统：孟加拉文、梵文、古吉拉特文、旁遮普文、埃纳德文、高棉文、马拉亚拉姆文、叙利亚文、泰米尔文、泰卢固文、塔安那文、迪维希文和藏文。最后，Qt 在 X11 上还支持奥里亚文，并且在 Windows XP 上还支持蒙古文和僧伽罗文。假设在系统上已经安装好了适当的字体，那么 Qt 就可以使用这些文字系统中的任何一种来显示文本。并且，假设已经安装好了适当的输入法，那么用户将可以在他们的 Qt 应用程序中使用这些文字系统输入文本了。

使用 QChar 编程和使用 char 编程有一点点不同。为了获得一个 QChar 的数字值，可以对它调用 unicode()。为了获得一个 QChar（作为一个 char）的 ASCII 或者 Latin-1 的值，则要对它调用 toLatin1()。对于那些非 Latin-1 的字符，toLatin1()将会返回“\0”。

如果我们知道一个程序中的所有字符都是 ASCII 字符，那么可以在 toLatin1()的返回值上使用像 isalpha()、isdigit()和 isspace()这样的标准 <cctype> 函数。然而，在执行这些操作时使用 QChar 的成员函数通常会是更好的方法，因为它们可用于任意的 Unicode 字符。QChar 提供的这些函数包括 isPrint()、isPunct()、isSpace()、isMark()、isLetter()、isNumber()、isLetterOrNumber()、isDigit()、isSymbol()、isLower()以及 isUpper()。例如，以下给出了测试一个字符是数字还是大写字符的一种方法：

```
if (ch.isDigit() || ch.isUpper())
    ...
```

这个代码片断可以用于区分任意的大写或者小写格式的字母，包括拉丁文、希腊文和西里尔文。

一旦有一个 Unicode 字符串，我们就可以在 Qt 的应用程序编程接口中任何需要 QString 的地方使用它。然后，就由 Qt 负责对其适当地加以显示，并且在和操作系统对话时负责把它转换成相应的编码格式。

当读取和写入文本文件时，需要给予特殊关注。文本文件可以使用很多种编码格式，并且通常很有可能需要从文本文件的内容来猜测它的编码格式。默认情况下，QTextStream 对于读取和写入都使用系统的本地 8 位编码格式[这可以通过 QTextCodec::codecForLocale()函数获得]。对于美国和西欧的本地设置，这种做法通常获得的就是 Latin-1。

如果要设计自己的文件格式，并且希望能够读取和写入任意的 Unicode 字符，那么可以在开始写入 QTextStream 之前调用以下代码：

```
stream.setCodec("UTF-16");
stream.setGenerateByteOrderMark(true);
```

这样就可以把数据保存为 Unicode 格式。然后,这些数据将会被保存为 UTF-16,这是一种每个字符都使用两个字节存储的格式,并且它会使用一个特殊的 16 位值(Unicode 字节序列掩码,0xFFFE)作为前缀,用于识别该文件采用的是 Unicode 编码,它也用来识别存储格式是以高字节在前还是高字节在后。UTF-16 格式和 QString 的内存表示形式一致,所以读取和写入 UTF-16 格式的 Unicode 字符串的速度非常快。另一方面,当我们把纯粹的 ASCII 数据保存为 UTF-16 格式的时候,总是会带来固有的开销,因为它存储每个字符的时候使用的是两个字节,而不是只用一个字节。

通过使用适当的 QTextCodec 来调用 setCodec(),还可以指定其他的编码方法。QTextCodec 是一个可以在 Unicode 和给定编码格式之间进行转换的对象。Qt 在很大范围的上下文关系上都使用了 QTextCodec。在 Qt 内部,把它们用于对字体、输入法、剪贴板、拖放和文件名的支持中。但是当我们编写 Qt 应用程序的时候,也可以使用它们。

当读取文本文件时,如果这个文件是以字节顺序标记(Byte Order Mark,BOM)开始的,那么 QTextStream 就会自动检测到 UTF-16。通过调用 setAutoDetectUnicode(false),可以关闭这一行为。如果假定数据不是以字节顺序标记开始的,那么在读取开始之前最好能够使用"UTF-16"先调用一下 setCodec()。

Unicode 完全支持的另外一种编码格式是 UTF-8。相对于 UTF-16,它的主要优点在于:它是 ASCII 的一个超集。位于 0x00 和 0x7F 范围之内的任何字符都被表示为一个单一字节。而其他字符,包括位于 0x7F 以上的 Latin-1 字符,则都使用多字节序列进行表示。对于以 ASCII 为主的文本,UTF-8 占用的空间约为 UTF-16 占用空间的一半。为了在 QTextStream 中使用 UTF-8,在读取和写入之前,可以以"UTF-8"作为编码格式的名字先调用一下 setCodec()。

如果我们总是希望忽略用户的本地设置而读取或者输出 Latin-1 字符,则可以把 QTextStream 的编码格式设置为"ISO 8859-1"。例如:

```
QTextStream in(&file);
in.setCodec("ISO 8859-1");
```

一些文件格式会在文件头指明它们所使用的编码格式。这个文件头通常是普通的 ASCII 编码,这样可以确保不管使用什么样的编码格式(这里我们假设它是 ASCII 的一个超集),都可以正确地读取这个文件。XML 文件格式就是这样一个有趣的例子。XML 文件通常编码为 UTF-8 或者 UTF-16。读取它们的适当方式是使用带"UTF-8"的 setCodec()调用。如果格式是 UTF-16,QTextStream 将会自动检测到该格式并且进行自我调整。XML 文件的 <?xml?> 样式的文件头有时会包含一个 encoding 参数,例如:

```
<?xml version="1.0" encoding="EUC-KR"?>
```

由于 QTextStream 不允许在开始读取字符之后再改变这个编码格式,因此,考虑到显式编码的问题,正确读取文件的方法应当是使用适当的编码格式[可以使用 QTextCodec::codecForName()函数获得]重新开始读取文件。对于 XML 的这种情况,通过使用 Qt 的 XML 类,就可以避免由自己处理这些编码格式的麻烦了,正如第 16 章中讲述的那样。

QTextCodec 的另外一个用法是用于指明在源代码中出现的字符串的编码方式。假设有这样一个例子,有一个日本的程序员团队正在编写一个主要面向日本家庭市场的应用程序。这些程序员喜欢使用编码方式为 EUC-JP 或者 Shift-JIS 的文本编辑器编写他们的源代码。这样的编辑器可以让他们方便地输入日文字符,所以,他们就可以输入像下面这样的程序代码:

```
QPushButton *button = new QPushButton(tr("日諾"));
```

默认情况下，Qt 会把 tr()的参数当作 Latin-1。为了改变这一点，可以调用静态函数 QTextCodec::setCodecForTr()。例如：

```
QTextCodec::setCodecForTr(QTextCodec::codecForName("EUC-JP"));
```

这个调用必须在第一次对 tr()的调用之前完成。通常，应当在 main()函数中来完成它，并且是在创建完 QApplication 对象之后就马上来做这项工作。

而在程序中给定的其他字符串仍将会当作 Latin-1 字符串。如果程序员想在以上情况下也输入日文字符，则可以使用 QTextCodec 明确地把它们转换成为 Unicode：

```
QString text = japaneseCodec->toUnicode("海鮮料理");
```

作为选择，当在 const char * 和 QString 之间进行转换的时候，可以通过调用 QTextCodec::setCodecForCStrings()来让 Qt 使用某种特殊的编解码器：

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForName("EUC-JP"));
```

以上所述的技术也同样适用于任意的非 Latin-1 语言，其中包括中文、希腊文、韩文和俄文。

以下列表给出了 Qt 4.3 所支持的编码方式：

- Apple Roman
- Big5
- Big5-HKSCS
- EUC-JP
- EUC-KR
- GB18030-0
- IBM 850
- IBM 866
- IBM 874
- ISO 2022-JP
- ISO 8859-1
- ISO 8859-2
- ISO 8859-3
- ISO 8859-4
- ISO 8859-5
- ISO 8859-6
- ISO 8859-7
- ISO 8859-8
- ISO 8859-9
- ISO 8859-10
- ISO 8859-13
- ISO 8859-14
- ISO 8859-15
- ISO 8859-16
- Iscii-Bng
- Iscii-Dev
- Iscii-Gir
- Iscii-Knd
- Iscii-Mlm
- Iscii-Ori
- Iscii-Pnj
- Iscii-Tlg
- Iscii-Tml
- JIS X 0201
- JIS X 0208
- KOI8-R
- KOI8-U
- MuleLao-1
- ROMAN8
- Shift-JIS
- TIS-620
- TSCII
- UTF-8
- UTF-16
- UTF-16BE
- UTF-16LE
- Windows-1250
- Windows-1251
- Windows-1252
- Windows-1253
- Windows-1254
- Windows-1255
- Windows-1256
- Windows-1257
- Windows-1258
- WINSAM12

对于列出的所有这些编码方式，QTextCodec::codecForName()总是可以返回一个有效的指针。通过对 QTextCodec 的子类化，也可以支持其他类型的编码方式。

18.2　让应用程序感知翻译

如果想让应用程序能使用多种语言，必须做两件事情：

- 确保每一个用户可见的字符串都使用了 tr()函数。
- 在应用程序启动的时候，载入了一个翻译文件(.qm)。

对于应用程序来说，以上两者都是必不可少的，否则就无法对其进行翻译。然而，使用 tr()通常并不需要付出太多的努力，并且这样做也为日后的翻译工作做好了准备。

tr()函数是定义在 QObject 中的一个静态函数,并且可以把它在任何一个定义了 Q_OBJECT 宏的子类中重写。在编写 QObject 子类的代码时,可以不拘泥于任何形式而直接调用 tr()。如果有可以使用的翻译,tr()调用就会返回其翻译;否则,就返回其原有的文本内容。在不是 QObject 的类中,可以使用带 QObject 类前缀的方式 QObject::tr()重写 tr()函数,也可以使用 Q_DECLARE_TR_FUNCTIONS()宏把 tr()函数添加到该类中,就像第 8 章所做的那样。

为了准备翻译文件,必须运行 Qt 的 lupdate 工具。这个工具会提取出现在 tr()调用中的所有字符串文字并且产生一些翻译文件,其中包含了所有这些准备要翻译的字符串。然后,这些文件就可以发送给某个翻译者,由其添加翻译内容。在 18.4 节中,将会说明这一过程。

tr()调用一般具有如下语法格式:

```
Context::tr(sourceText, comment)
```

这里的 *Context* 部分就是使用 Q_OBJECT 宏定义的 QObject 子类的名字。如果我们从正在使用的类的一个成员函数中调用了 tr()函数,那么就不需要再指定这个 *Context* 部分。*sourceText* 部分是需要翻译的字符串文字。*comment* 部分是可选的,它可以用来向翻译者提供一些额外的信息。

下面是一些例子:

```
RockyWidget::RockyWidget(QWidget *parent)
    : QWidget(parent)
{
    QString str1 = tr("Letter");
    QString str2 = RockyWidget::tr("Letter");
    QString str3 = SnazzyDialog::tr("Letter");
    QString str4 = SnazzyDialog::tr("Letter", "US paper size");
}
```

在前面两个 tr()调用中,都是以"RockyWidget"作为它们的上下文,而在后面的两个 tr()调用中,则以"SnazzyDialog"作为上下文。这 4 个 tr()调用中,全部都是以"Letter"作为源文本。最后一个调用还有一个备注说明,用来帮助翻译人员理解源文本所要表达的意思。

在不同上下文(类)中的字符串可以相互独立地翻译出来。翻译者通常每次只在一个上下文中进行翻译工作,通常会让应用程序运行并且显示要翻译的窗口部件或者对话框。

当从全局函数中调用 tr()时,必须明确地指定上下文。在应用程序中,任何 QObject 的子类都可以用作上下文。如果没有一个合适,那么总是可以使用 QObject 自身来当作上下文。例如:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ...
    QPushButton button(QObject::tr("Hello Qt!"));
    button.show();
    return app.exec();
}
```

在目前为止的每一个例子中,上下文曾经都有一个类名。这是非常方便的,因为我们几乎可以总是忽略它。但是,并非所有的情况总是如此。在 Qt 中,翻译一个字符串最常用的方法是使用 QCoreApplication::translate()函数,它可以最多接受三个参数:上下文、源文本以及可选的注释项。例如,以下是翻译"Hello Qt!"的另外一种方式:

```
QCoreApplication::translate("Global Stuff", "Hello Qt!")
```

这一次,我们把要翻译的文本放到了"Global Stuff"的上下文中。

tr()和 translate()这两个函数都有一个双重用途:它们既都是 lupdate 用来找到用户可见字符串的标记符,同时又都是可以翻译文本的 C++ 函数。因此,这样对我们如何编写代码就产生了一些

影响。例如,下面的代码就将无法工作:

```
// 错误
const char *appName = "OpenDrawer 2D";
QString translated = tr(appName);
```

这里的问题就在于 lupdate 不能提取这个“OpenDrawer 2D”字符串文字,因为它没有出现在 tr()函数的调用中。这也就意味着翻译者将没有机会翻译这个字符串。在与动态字符串组合使用时,也会引起这个问题:

```
// 错误
statusBar()->showMessage(tr("Host " + hostName + " found"));
```

在这个例子中,传递给 tr()的字符串会根据 hostName 的值的不同而不同,所以我们就不能想当然地期望 tr()函数可以正确地翻译它。

这一问题的解决方法是使用 QString::arg():

```
statusBar()->showMessage(tr("Host %1 found").arg(hostName));
```

注意它是如何工作的:传递给 tr()的字符串文本是“Host %1 found”。假设加载的是一个法语翻译文件,那么 tr()将会返回一个像“Hôte %1 trouvé”这样的字符串。然后,再使用 hostName 变量中的内容替换“%1”这个参数。

尽管对一个变量调用 tr()通常是非常不明智的事情,但的确可以让它正常工作。在把一个字符串文字赋值给一个变量之前,为了能够翻译它,必须使用 QT_TR_NOOP()宏先标记它。对于静态字符串数组来说,这样做是非常有用的。例如:

```
void OrderForm::init()
{
    static const char * const flowers[] = {
        QT_TR_NOOP("Medium Stem Pink Roses"),
        QT_TR_NOOP("One Dozen Boxed Roses"),
        QT_TR_NOOP("Calypso Orchid"),
        QT_TR_NOOP("Dried Red Rose Bouquet"),
        QT_TR_NOOP("Mixed Peonies Bouquet"),
        0
    };

    for (int i = 0; flowers[i]; ++i)
        comboBox->addItem(tr(flowers[i]));
}
```

QT_TR_NOOP()宏只是简单地返回它的参数。但是 lupdate 将会提取所有包含在 QT_TR_NOOP()中的字符串,这样就可以翻译它们了。当在以后使用这个变量时,就可以像平常那来调用 tr()来执行翻译。尽管已经给 tr()传递了一个变量,但是翻译仍将可以继续执行。

还有一个 QT_TRANSLATE_NOOP()宏,它的工作方式就像 QT_TR_NOOP()宏一样,只是它还带有一个上下文作为参数。当在类之外初始化它的变量时,这个宏就会显得非常有用:

```
static const char * const flowers[] = {
    QT_TRANSLATE_NOOP("OrderForm", "Medium Stem Pink Roses"),
    QT_TRANSLATE_NOOP("OrderForm", "One Dozen Boxed Roses"),
    QT_TRANSLATE_NOOP("OrderForm", "Calypso Orchid"),
    QT_TRANSLATE_NOOP("OrderForm", "Dried Red Rose Bouquet"),
    QT_TRANSLATE_NOOP("OrderForm", "Mixed Peonies Bouquet"),
    0
};
```

上下文参数必须与在后面的 tr()或者 translate()中所使用的上下文一致。

当在应用程序中使用 tr()时,非常容易忘记对一些用户可见的字符串调用 tr(),特别是在刚开始使用它的时候。这些遗漏的 tr()调用最终可能会由翻译人员发现,或者甚至更糟,最终可能会由使用这个翻译过的应用程序的用户发现,因为这样的一些字符串是按照最初开发时的语言形式显示出来的。为了避免这个问题,可以告诉 Qt 禁止从 const char * 到 QString 的隐含转换。通过在包含任意的 Qt 头文件之前预先定义 QT_NO_CAST_FROM_ASCII 预处理器符号,就可以实现这一点。要确保设置过这个符号的最为简单的方式就是在应用程序的 .pro 文件中添加如下一行代码:

```
DEFINES += QT_NO_CAST_FROM_ASCII
```

这样将会强制每一个字符串文字都需要使用 tr()或者 QLatin1String()的封装,这取决于它们是否应当翻译。没有正确封装的字符串将会在编译时产生错误,从而会强迫我们添加那些被遗漏的 tr()或者 QLatin1String()调用。

一旦对所有用户可见的字符串全部使用了 tr()调用封装,那么要使翻译生效,剩下的最后一件事情就是加载翻译文件。通常情况下,应当在应用程序的 main()函数中完成这一步操作。例如,下面给出了根据用户的本地设置,我们应当如何试着加载一个翻译文件的代码:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ...
    QTranslator appTranslator;
    appTranslator.load("myapp_" + QLocale::system().name(), qmPath);
    app.installTranslator(&appTranslator);
    ...
    return app.exec();
}
```

这里的 QLocale::system()函数会返回一个 QLocale 对象,用于提供用户的本地设置信息。习惯上,我们使用本地的名称作为 .qm 文件名的一部分。本地名称可以是更精确的,也可以是不太精确的;例如,fr 可以用来说明一个法语的本地设置,fr_CA 则可以用来说明一个法裔加拿大人的本地设置,fr_CA.ISO8859-15 则可以用来说明一个使用 ISO 8859-15 编码方式(一种可以支持"€"、"Œ"、"œ"和"Ÿ"的编码方式)的法裔加拿大人的本地设置。

假设本地设置是 fr_CA.ISO8859-15,则 QTranslator::load()函数会首先试着加载 myapp_fr_CA.ISO8859-15.qm 文件。如果这个文件不存在,那么接下来 load()会尝试加载 myapp_fr_CA.qm,然后再尝试加载 myapp_fr.qm,并且在最后放弃之前再试试 myapp.qm 文件。通常,我们应当只提供一个 myapp_fr.qm,其中包含着标准的法语翻译,但是如果需要为说法语的加拿大人提供一个不同的文件,则也可以提供一个 myapp_fr_CA.qm 文件,那么它将会应用到 fr_CA 的本地设置中。

QTranslator::load()中的第二个参数表示我们希望 load()到哪个目录中查找翻译文件的路径。在这个例子中,我们假设这些翻译文件位于变量 qmPath 给定的路径中。

在 Qt 库中,包含了一些需要翻译的字符串。Trolltech 在 Qt 的 translations 目录下提供了法语、德语和简体中文的翻译文件。它还提供了其他一些翻译语言,但它们是由 Qt 用户所贡献的,因而官方对它们不予支持。也可以使用如下方式加载 Qt 库中的翻译文件:

```
QTranslator qtTranslator;
qtTranslator.load("qt_" + QLocale::system().name(), qmPath);
app.installTranslator(&qtTranslator);
```

由于一个 QTranslator 对象每次只能保存一个翻译文件,所以我们为 Qt 的翻译使用了一个单独的 QTranslator 对象。让每一个翻译人员只使用一个文件不会是什么问题,因为可以按照需要安装足够多的翻译文件。当 QCoreApplication 要查找翻译文件的时候,它将会用到所有这些翻译文件。

一些文字，诸如阿拉伯文和希伯来文，它们是从右向左而不是从左向右书写的。对于它们，必须颠倒应用程序的整个布局，这可以通过调用 QApplication::setLayoutDirection (Qt::RightToLeft)来完成。用于 Qt 的翻译文件中包含了一个名为“LTR”的特殊标记，它可以告诉 Qt 这个语言的书写形式是从左向右还是从右向左，所以通常并不需要我们亲自调用 setLayoutDirection()。

如果通过 Qt 的资源系统把翻译文件嵌入到可执行文件中来改变应用程序，那么也许可以为用户提供更多的方便。此外，这样做不仅减少了作为发布产品一部分的文件数量，而且也避免了翻译文件丢失或者被意外删除的风险。假设 .qm 文件放在资源树中的 translations 子目录中，于是我们就应当有一个 myapp.qrc 文件，它具有如下所示的内容：

```
<RCC>
<qresource>
    <file>translations/myapp_de.qm</file>
    <file>translations/myapp_fr.qm</file>
    <file>translations/myapp_zh.qm</file>
    <file>translations/qt_de.qm</file>
    <file>translations/qt_fr.qm</file>
    <file>translations/qt_zh.qm</file>
</qresource>
</RCC>
```

.pro 文件将会包含如下一条：

```
RESOURCES += myapp.qrc
```

最后，在 main()中，必须指明“:/translations”作为翻译文件的路径。前面的冒号说明，指向资源文件的路径与文件系统中指向文件的路径是不同的。

现在，我们已经讲述了如何让应用程序能够使用翻译文件以其他不同的语言进行操作所需的全部内容。但是，国与国和文化与文化之间的区别，不仅只表现在语言和文字系统的方向上的不同。国际化的程序还必须考虑使用本地的日期和时间格式、货币格式、数字格式以及字符串的结合顺序。Qt 包含了一个 QLocale 类，它可以用来提供本地化的数字和日期以及时间格式。要查询其他与本地相关的特定信息，可以使用标准 C++ 中的 setlocale()和 localeconv()函数。

Qt 中的一些类和函数可以根据这些本地设置来调整它们的行为：

- QString::localeAwareCompare()使用与本地设置相关的方式比较两个字符串。在对用户的可见项进行排序的时候，它非常有用。
- 由 QDate、QTime 和 QDateTime 提供的 toString()函数，当使用 Qt::LocalDate 作为参数而调用它的时候，这个函数可以返回一个具有本地格式的字符串。
- 默认情况下，QDateEdit 和 QDateTimeEdit 窗口部件都使用本地格式来显示日期和时间。

最后，一个经过翻译的应用程序可能在某些特定的情况下需要使用和原有图标不同的图标。例如，Web 浏览器上的 Back 按钮和 Forward 按钮，在处理一种从右向左的语言的时，它们上面的左右箭头就应该互换一下。可以使用如下的方法来完成这一工作：

```
if (QApplication::isRightToLeft()) {
    backAction->setIcon(forwardIcon);
    forwardAction->setIcon(backIcon);
} else {
    backAction->setIcon(backIcon);
    forwardAction->setIcon(forwardIcon);
}
```

通常情况下，对于包含字母字符的所有图标都需要翻译。例如，与文字处理软件中的倾斜(Italic)字体选项关联在一起的工具条按钮上的字符“I”，在西班牙语中就应当替换为“C”(Cursivo)，

并且在丹麦语、荷兰语、德语、挪威语和瑞典语中就应该替换为“K”(Kursiv)。以下给出了实现这一点的简单做法：

```
if (tr("Italic")[0] == 'C') {
    italicAction->setIcon(iconC);
} else if (tr("Italic")[0] == 'K') {
    italicAction->setIcon(iconK);
} else {
    italicAction->setIcon(iconI);
}
```

还有另外一种方法，就是利用资源系统对多个本地设置提供支持的特性。在 .qrc 文件中，我们可以使用 lang 属性为一种资源指定一个本地化设置。例如：

```
<qresource>
    <file>italic.png</file>
</qresource>
<qresource lang="es">
    <file alias="italic.png">cursivo.png</file>
</qresource>
<qresource lang="sv">
    <file alias="italic.png">kursiv.png</file>
</qresource>
```

如果用户的本地设置是 es(Español)，那么，“:/italic.png”就成为 cursivo.png 的图片引用。如果本地设置是 sv(Svenska)，那么就会使用 kursiv.png 图片。对于其他情况，就会使用 italic.png。

18.3 动态切换语言

对于绝大多数应用程序，在 main()中检测用户的首选语言并为之加载适当的 .qm 文件，能够这样做是非常完美的事情。但是在一些情况下，用户也许需要程序具有动态切换语言的功能。一个由不同人不间断地轮换使用的应用程序，也许就非常需要能够在不重新启动应用程序的情况下改变语言。例如，由呼叫中心操作员、同声翻译人员或者采用计算机系统的收银机操作人员使用的应用程序通常就需要具备这种能力。

让应用程序能够动态切换语言，需要比在开始处只加载一个单一的翻译文件要稍微多做一点工作，但是这并不是什么困难的事情。以下是必须完成的工作：

- 提供用户可以用来切换语言的一种方法。
- 对于每一个窗口部件或者对话框，把它所有可翻译的字符串放在一个单独的函数[通常称为 retranslateUi()]中，并且当语言发生改变的时候调用这个函数。

下面讨论与呼叫中心应用程序相关部分的源代码。这个应用程序提供了一个 Language 菜单(如图 18.1 所示)，允许用户在程序运行时设置它的语言。默认的语言是英语。



图 18.1 动态的 Language 菜单

由于不知道应用程序在启动的时候用户将会使用哪一种语言，所以不必在 main()函数中加载这些翻译文件。相反，当需要这些翻译文件的时候，我们将会动态地加载它们。所以，需要用来处理翻译的所有代码必须放在这些主窗口和对话框类的里面。

让我们来看看这个呼叫中心应用程序的 QMainWindow 子类。

```
MainWindow::MainWindow()
{
```

```
    journalView = new JournalView;
    setCentralWidget(journalView);

    qApp->installTranslator(&appTranslator);
    qApp->installTranslator(&qtTranslator);

    createActions();
    createMenus();

    retranslateUi();
}
```

在这个构造函数中，我们把 JournalView 设置为中央窗口部件，它是 QTableWidget 的一个子类。然后，在 QApplication 上安装了两个 QTranslator 对象：appTranslator 对象存储应用程序的当前翻译，qtTranslator 对象存储 Qt 的翻译。最后，调用 createActions()和 createMenus()这两个私有函数来创建菜单系统，并且调用 retranslateUi()(也是一个私有函数)设置程序在第一次运行时用户可见的那些字符串。

```
void MainWindow::createActions()
{
    newAction = new QAction(this);
    newAction->setShortcut(QKeySequence::New);
    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
    ...
    exitAction = new QAction(this);
    connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
    ...
    aboutQtAction = new QAction(this);
    connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}
```

createActions()函数像通常那样创建了一些 QAction 对象，但是它没有设置任何文本。这些工作将会在 retranslateUi()中完成。对于具有标准化快捷键的那些动作，可以在这里使用适当的项设置它们的快捷键，并在必要时依靠 Qt 来翻译。对于那些已经有自定义快捷键的动作，比如 Exit 动作，可以在 retranslateUi()函数中与它的文本一起设置它的快捷键。

```
void MainWindow::createMenus()
{
    fileMenu = new QMenu(this);
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(exitAction);

    editMenu = new QMenu(this);
    ...
    createLanguageMenu();

    helpMenu = new QMenu(this);
    helpMenu->addAction(aboutAction);
    helpMenu->addAction(aboutQtAction);

    menuBar()->addMenu(fileMenu);
    menuBar()->addMenu(editMenu);
    menuBar()->addMenu(reportsMenu);
    menuBar()->addMenu(languageMenu);
    menuBar()->addMenu(helpMenu);
}
```

createMenus()函数创建菜单，但是没有向菜单条中插入任何标题。再次重申一下，这些工作将会在 retranslateUi()中完成。

在该函数的中间部分，我们调用 createLanguageMenu()来使用所支持的语言列表填充 Language

菜单。稍后将会看到这部分源代码。在此之前,先来看一下 retranslateUi()函数:

```
void MainWindow::retranslateUi()
{
    newAction->setText(tr("&New"));
    newAction->setStatusTip(tr("Create a new journal"));
    ...
    exitAction->setText(tr("E&xit"));
    exitAction->setShortcut(tr("Ctrl+Q"));
    ...
    aboutQtAction->setText(tr("About &Qt"));
    aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));

    fileMenu->setTitle(tr("&File"));
    editMenu->setTitle(tr("&Edit"));
    reportsMenu->setTitle(tr("&Reports"));
    languageMenu->setTitle(tr("&Language"));
    helpMenu->setTitle(tr("&Help"));

    setWindowTitle(tr("Call Center"));
}
```

这个 retranslateUi()函数就是 MainWindow 类中所有出现 tr()调用的地方。它是在 MainWindow 构造函数的最后部分得到调用的,并且在用户每次使用 Language 菜单改变应用程序语言的时候也会调用这个函数。

我们为每一个 QAction 设置其文本和状态提示,并且为那些具有非标准快捷键的动作设置其快捷键。我们还为每一个 QMenu 设置它们的标题,也要设置窗口的标题。

在前面提到的 createMenus()函数调用 createLanguageMenu(),用来弹出一个带有语言列表的 Language 菜单:

```
void MainWindow::createLanguageMenu()
{
    languageMenu = new QMenu(this);

    languageActionGroup = new QActionGroup(this);
    connect(languageActionGroup, SIGNAL(triggered(QAction *)),
            this, SLOT(switchLanguage(QAction *)));

    QDir qmDir = directoryOf("translations");
    QStringList fileNames =
            qmDir.entryList(QStringList("callcenter_*.qm"));

    for (int i = 0; i < fileNames.size(); ++i) {
        QString locale = fileNames[i];
        locale.remove(0, locale.indexOf('_') + 1);
        locale.chop(3);

        QTranslator translator;
        translator.load(fileNames[i], qmDir.absolutePath());
        QString language = translator.translate("MainWindow",
                                                "English");

        QAction *action = new QAction(tr("&%1 %2")
                                      .arg(i + 1).arg(language), this);
        action->setCheckable(true);
        action->setData(locale);

        languageMenu->addAction(action);
        languageActionGroup->addAction(action);

        if (language == "English")
            action->setChecked(true);
    }
}
```

和在应用程序中通过硬编码来支持多种语言有所不同，我们在应用程序的 translations 目录下为每一个 .qm 文件创建一个菜单项。directoryOf()函数就是第 17 章中用过的那个函数。

为简单起见，我们假设英文(English)也有一个 .qm 文件。当用户选择 English 的时候，还有一种方法本应当对这些 QTranslator 对象调用 clear()。

这里存在一个特殊的困难，就是希望用一个漂亮的名字来表示每一个 .qm 文件所能提供的语言。如果只是基于每一个 .qm 文件的名字，即只为“English”显示一个“en”，或者只为“Deutsch”显示一个“de”，这样的命名方法看起来就会显得有些过于简练，并且容易让用户产生困惑。在 createLanguageMenu()中使用的解决方法就是在“MainWindow”的上下文中检查“English”这个字符串的翻译。这个字符串在德语的翻译中就应该被翻译为“Deutsch”，而在一个法语翻译文件中就应该翻译为“Français”，在一个日语翻译文件中就应当翻译成“日本語”。

我们为每一种语言都创建一个可复选型的 QAction，并且把这个本地名称保存在这个动作的“data”项中。我们把它们添加到一个 QActionGroup 对象中，以确保每次只能选中 Language 菜单中的一项。当用户从这个群组中选择一个动作时，QActionGroup 就会发射一个 triggered(QAction *)信号，该信号则连接到了 switchLanguage()。

```
void MainWindow::switchLanguage(QAction *action)
{
    QString locale = action->data().toString();
    QString qmPath = directoryOf("translations").absolutePath();

    appTranslator.load("callcenter_" + locale, qmPath);
    qtTranslator.load("qt_" + locale, qmPath);
    retranslateUi();
}
```

当用户从 Language 菜单中选择一种语言的时候，就会调用 switchLanguage()槽。我们可以为应用程序和 Qt 加载翻译文件，并且可以调用 retranslateUi()，由其为主窗口重新翻译所有的字符串。

在 Windows 系统上，还有一种方法提供对 LocaleChange 事件做出响应的 Language 菜单，当 Qt 检测到环境的本地设置发生变化时，它就会发射一个这种类型的事件。Qt 在各个平台上都支持这种类型的事件，但是实际上只有在 Windows 上，当用户改变系统的本地设置的时候，才会产生这种事件(在控制面板中的“地区和语言”选项中)。为了处理 LocaleChange 事件，可以像下面这样重新实现 QWidget::changeEvent()：

```
void MainWindow::changeEvent(QEvent *event)
{
    if (event->type() == QEvent::LocaleChange) {
        QString qmPath = directoryOf("translations").absolutePath();
        appTranslator.load("callcenter_"
                           + QLocale::system().name(), qmPath);

        qtTranslator.load("qt_" + QLocale::system().name(), qmPath);
        retranslateUi();
    }
    QMainWindow::changeEvent(event);
}
```

当应用程序正在运行的时候，如果用户切换了本地设置，那么就会试图为新的本地设置加载正确的翻译文件并且调用 retranslateUi()以更新用户界面。无论在何种情况下，都要把这个事件传递给基类的 changeEvent()函数，因为这个基类也许对 LocaleChange 或者其他发生变化的事件感兴趣。

现在，已经完成了对 MainWindow 代码的回顾。下一步，将查看应用程序中的一个窗口部件类——JournalView 类——的代码，来看看我们为了让它支持动态翻译需要做哪些改变。

```
JournalView::JournalView(QWidget *parent)
    : QTableWidget(parent)
{
    ...
    retranslateUi();
}
```

JournalView 类是 QTableWidget 的一个子类。在该构造函数的最后,我们调用 retranslateUi()私有函数来设置该窗口部件的字符串。这与 MainWindow 中所做过的非常相似。

```
void JournalView::changeEvent(QEvent *event)
{
    if (event->type() == QEvent::LanguageChange)
        retranslateUi();
    QTableWidget::changeEvent(event);
}
```

我们还重新实现了 changeEvent()函数,对 LanguageChange 事件调用 retranslateUi()函数。在安装到 QCoreApplication 中的当前 QTranslator 的内容发生变化时,Qt 就会产生一个 LanguageChange 事件。在应用程序中,当在 MainWindow::switchLanguage()或者在 MainWindow::changeEvent()中对 appTranslator 或者 qtTranslator 调用 load()时,就会发生这种情况。

不应当混淆 LanguageChange 事件和 LocaleChange 事件。LocaleChange 事件由系统产生并且会告诉应用程序:"也许应当加载一个新的翻译文件了"。LanguageChange 事件则是由 Qt 产生的,它会告诉应用程序的窗口部件:"也许应当重新翻译所有字符串了"。

当重新实现 MainWindow 时,不需要对 LanguageChange 做出响应。相反,当对 QTranslator 调用 load()时,只需简单地调用 retranslateUi()就可以了。

```
void JournalView::retranslateUi()
{
    QStringList labels;
    labels << tr("Time") << tr("Priority") << tr("Phone Number")
           << tr("Subject");
    setHorizontalHeaderLabels(labels);
}
```

retranslateUi()函数使用新翻译的文本更新列的首部,完成手写代码的窗口部件中与翻译相关的代码。

18.4 翻译应用程序

翻译一个含有 tr()调用的 Qt 应用程序就是一个由三步构成的过程:

1. 运行 lupdate,从应用程序的源代码中提取所有用户可见的字符串。
2. 使用 Qt Linguist 翻译该应用程序。
3. 运行 lrelease,生成二进制的 .qm 文件,应用程序可以使用 QTranslator 加载这个文件。

第 1 步和第 3 步由应用程序开发人员执行,第 2 步由翻译人员处理。根据这个应用程序的开发过程和使用的生命周期的需要,可以多次重复这一循环过程。

作为一个例子,我们将会演示如何翻译第 3 章中的 Spreadsheet 应用程序。在这个应用程序中,已经对每一个用户可见的文本都包含了 tr()调用。

首先,必须稍微修改一下应用程序的 .pro 文件,以便说明我们想要支持哪些语言。例如,如果想除了支持英语之外,还想支持德语和法语,则需要在 spreadsheet.pro 文件中添加如下的 TRANSLATIONS 项:

```
TRANSLATIONS = spreadsheet_de.ts \
               spreadsheet_fr.ts
```

这里给定了两个翻译文件:一个用于德语,另一个用于法语。当第一次运行 lupdate 时,将会创建这两个文件,并且在以后每次执行 lupdate 的时候,都会更新这两个文件。

这些文件通常都有一个 .ts 扩展名。它们都使用简单的 XML 格式,并且都没有像可由 QTranslator 理解的二进制 .qm 文件那样紧凑。把人们可读的 .ts 文件转换成高效率的机器可理解的 .qm 文件则是 lrelease 的工作。大家不用好奇,.ts 文件的名字代表“翻译源”(translation source),而 .qm 文件的名字代表“Qt 消息”(Qt message)。

假设我们现在位于含有 Spreadsheet 应用程序源代码的目录中,那么就可以在命令行下执行下面的命令,以用来在 spreadsheet.pro 中运行 lupdate:

```
lupdate -verbose spreadsheet.pro
```

这里的-verbose 选项告诉 lupdate 要比平常提供更多的反馈信息。以下给出的是预期的输出结果:

```
Updating 'spreadsheet_de.ts'...
    Found 98 source texts (98 new and 0 already existing)
Updating 'spreadsheet_fr.ts'...
    Found 98 source texts (98 new and 0 already existing)
```

在应用程序源代码的每个 tr()调用中所出现的每一个字符串,都会存储到这些 .ts 文件中,后面会跟一个空白翻译。应用程序的 .ui 文件中所出现的字符串也会包含在其中。

默认情况下,lupdate 工具会假设 tr()中的参数都是 Latin-1 字符串。如果不是这种情况,那么就必须在 .pro 文件中添加一个 CODECFORTR 项。例如:

```
CODECFORTR = EUC-JP
```

除了在应用程序的 main()函数中调用 QTextCodec::setCodecForTr()之外,还必须完成这一点。

然后,就需要使用 Qt Linguist 把这些翻译添加到 spreadsheet_de.ts 和 spreadsheet_fr.ts 文件中。(图 18.2 给出了使用中的 Qt Linguist。)

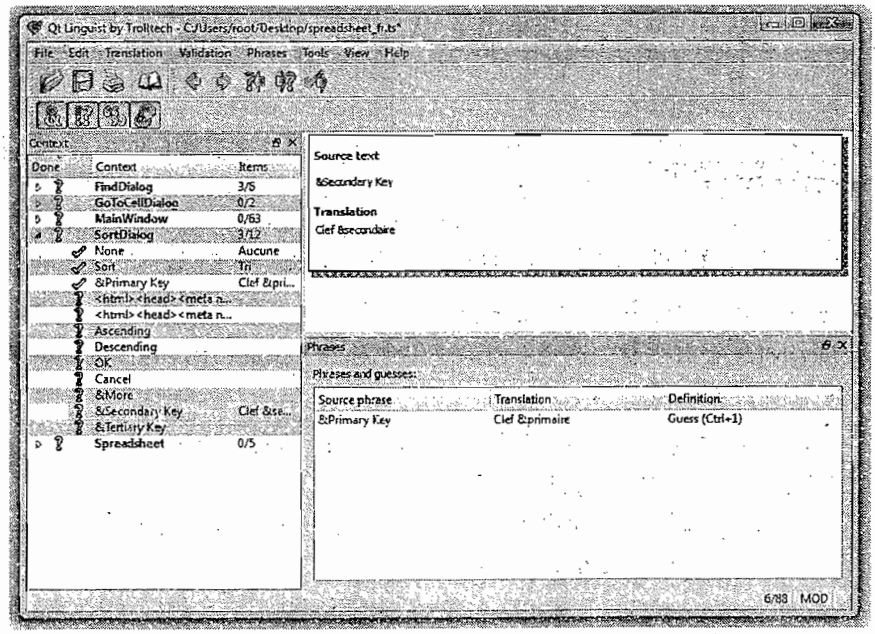


图 18.2　运行中的 Qt Linguist

要运行 Qt Linguist,在 Windows 上,可以在"开始"菜单中单击"Qt by Trolltech v4.x.y"→"Linguist";在 UNIX 上,可以在命令行中键入 linguist 命令;在 Mac OS X 里,可在 Finder 中双击 Linguist。要在一个 .ts 文件中添加翻译文本,可以单击 File→Open 并且选择需要翻译的文件即可。

在 Qt Linguist 主窗口的左侧一栏,显示的是一个树状视图。那些顶层项是要翻译的应用程序上下文。对于 Spreadsheet 应用程序,这些上下文是"FindDialog"、"GoToCellDialog"、"MainWindow"、"SortDialog"和"Spreadsheet"。每一个上下文都有两个或者更多的子项。每一个子项都会占用三列,第一列显示一个 Done 标记,第二列显示一个源文本,第三列显示任意的翻译内容。右上区域显示的是当前源文本以及它的翻译,这就是添加翻译和编辑翻译的地方。右下区域是一个由 Qt Linguist 自动提供的建议列表。

一旦有了一个翻译过的 .ts 文件,就需要把它转换成一个可以由 QTranslator 使用的二进制 .qm 文件。为了在 Qt Linguist 中完成这一点,可以单击 File→Release。通常情况下,应当先只翻译几个字符串,并且让应用程序在运行的时候使用该 .qm 文件来确保一切都可以正常工作。

如果想为所有的 .ts 文件重新生成这些 .qm 文件,可以采用下面的方法来使用 lrelease 工具:

```
lrelease -verbose spreadsheet.pro
```

假设我们把 19 个字符串翻译成了法语,并且为它们中的 17 个单击了 Done 标记,那么 lrelease 将会产生下面这样的输出:

```
Updating 'spreadsheet_de.qm'...
    Generated 0 translations (0 finished and 0 unfinished)
    Ignored 98 untranslated source texts
Updating 'spreadsheet_fr.qm'...
    Generated 19 translations (17 finished and 2 unfinished)
    Ignored 79 untranslated source texts
```

当应用程序运行的时候,还没有被翻译的字符串将会按最初的语言显示出来。lrelease 将会忽略 Done 标记,翻译人员可以用它来判断哪些翻译已经完成了,同时哪些还必须重新访问。

当修改应用程序源代码的时候,翻译文件就可能过时了。解决这个问题的方法就是再次运行 lupdate,这样就可以为新的字符串重新提供翻译,然后再重新生成这些 .qm 文件。一些开发团队发现经常运行 lupdate 是非常有用的,而其他一些开发团队则喜欢等到最终产品即将发布时才运行它。

lupdate 和 Qt Linguist 这些工具都非常智能。那些不再使用的翻译会仍旧保存在 .ts 文件中,因为很可能会在以后的发布中用到它们。在更新 .ts 文件的时候,lupdate 会使用一种智能的合并算法,它可以为翻译人员在处理那些不同上下文中具有相同或者相近文本的翻译时节省相当多的时间。

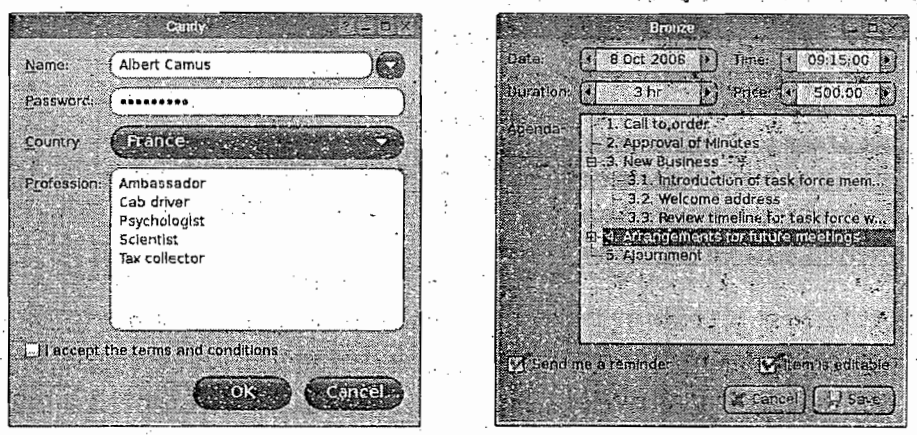
有关 Qt Linguist、lupdate 和 lrelease 的更多信息,请参考 http://doc.trolltech.com/4.3/ linguist-manual.html 中 Qt Linguist 的在线手册。这个手册除了包含对 Qt Linguist 用户界面的充分解释外,还为程序员准备了一份详细的教程。

第 19 章　自定义外观

在某些情况下，我们可能需要修改 Qt 内置窗口部件的外观。可能只希望稍微做一些美学方面的优化，或者希望实现一种全新的风格，以便可以为应用程序或者一批应用程序实现统一的、与众不同的外观。无论是哪种情况，都可以使用三种方法来重新定义 Qt 内置窗口部件的外观：

- 子类化个别的窗口部件类，并且重新实现它的绘制和鼠标事件处理器。这给我们以完全的可控性，但需要付出大量的工作。这意味着必须遍历所有的代码和 Qt 设计师的窗体，把 Qt 所有相关的类都改成子类。
- 子类化 QStyle 或者一个预定义的风格，例如 QWindowStyle。这种方法很好用，Qt 本身就是用这种方法为它所支持的不同平台提供基于平台的外观的。
- 从 Qt 4.2 开始，可以使用 Qt 样式表，这是一种从 HTML CSS(层叠样式表)获得灵感的机制。因为样式表是一种在运行时解释的普通文本文件，使用它们不需要具备任何编程知识。

第 5 章和第 7 章已经讲过第一种方法所需要的技术，然而我们强调的是自定义窗口部件。本章将介绍后两种方法。我们将展示两种自定义风格：一种是 Candy 风格，它通过样式表定义；另一种是 Bronze 风格，它通过 QStyle 的子类实现(如图 19.1 所示)。为了尽可能减小例子的大小，我们只选择一部分 Qt 窗口部件做介绍。



Candy　　Bronze

图 19.1　本章介绍的两种自定义样式

19.1　使用 Qt 样式表

Qt 的样式表主要是受到了 CSS 的启发，但同时也适用于窗口部件。样式表由影响窗口部件绘制的样式规则组成。这些规则是普通文本。由于在运行时会解析样式表，所以可以通过制定样式表的方式来尝试设计不同的 Qt 应用程序：使用-stylesheet *file.qss* 命令行参数、使用 Qt 设计师的样式表编辑器，或者在开发应用程序时嵌入一个 QTextEdit。

样式表作用于上层的当前激活的 QStyle 上(例如,QWindowsVistaStyle 或者 QPlastiqueStyle)①。因为创建样式表不引入任何子类,所以它们适合对现有窗口部件做微小的定制。例如:假设我们想在应用程序中的所有 QLineEdit 中使用黄色作为背景色,可以通过使用下面的样式表完成:

```
QLineEdit {
    background-color: yellow;
}
```

按照 CSS 的说法,QLineEdit 可以称为选择器(selector),background-color 就是属性,而 yellow 就是值。

对于这样的定制,使用样式表比修改窗口部件的调色板更可靠些。这是因为 QPalette 中的实体(Base、Button、Highlight,等等)在不同样式中的用法是不一样的。例如,QWindowsStyle 用 Base 调色板实体填充只读组合框的背景,然而,QPlastqueStyle 使用 Button 实体完成这一功能。此外,有些样式使用硬编码的图像呈现某个元素,从而跳过了调色板。相反,样式表则保证了无论激活的是哪种样式,都可以使用指定的颜色。

QApplication::setStyleSheet()为整个应用程序设置一个样式表:

```
qApp->setStyleSheet("QLineEdit { background-color: yellow; }");
```

还可以使用 QWidget::setStyleSheet()设置窗口部件以及其子窗口部件的样式表。例如:

```
dialog->setStyleSheet("QLineEdit { background-color: yellow; }");
```

如果直接在 QLineEdit 上设置样式表,可以忽略 QLineEdit 选择器以及大括号:

```
lineEdit->setStyleSheet("background-color: yellow;");
```

到目前为止,我们只是在单独一个类上设置了一个属性。在实践中,样式规则通常是组合的。例如,下面的规则设置 6 个窗口部件类以及它们的子类的前景色和背景色:

```
QCheckBox, QComboBox, QLineEdit, QListView, QRadioButton, QSpinBox {
    color: #050505;
    background-color: yellow;
}
```

颜色可以由名称、*#RRGGBB* 格式的 HTML 样式的字符串、RGB 或 RGBA 值指定:

```
QLineEdit {
    color: rgb(0, 88, 152);
    background-color: rgba(97%, 80%, 9%, 50%);
}
```

当使用颜色名称时,可以使用任何能被 QColor::setNameColor()函数识别的名字。对于 RGB 格式,必须指定红、绿、蓝部分,它们的取值范围是 0 至 255,或者 0%至 100%区间的值。RGBA 额外需要指定一个透明度值作为颜色的第四部分,它对应着颜色的不透明性。除了统一的颜色,也可以指定调色板实体或者一个渐变:

```
QLineEdit {
    color: palette(Base);
    background-color: qlineargradient(x1: 0, y1: 0, x2: 1, y2: 1,
                                      stop: 0 white, stop: 0.4 gray,
                                      stop: 1 green);
}
```

第 8 章介绍的三种渐变类型对应着 qlineargradient()、qradialgradient()以及 qconicalgradient()。语法在样式表的参考文档上有介绍。

使用 background-image 属性,可以指定一个图片作为背景:

① 在 Qt 4.3 中,样式表暂时还不支持 QMacStyle,但有望在以后的版本中得到完善。

```
QLineEdit {
    color: rgb(0, 88, 152);
    background-image: url(:/images/yellow-bg.png);
}
```

默认情况下，背景图片从窗口部件的左上角（不包含使用 margin 指定的边缘区域）开始，并且向水平和竖直方向重复填充整个窗口部件。这可以通过使用 background-position 和 background-repeat 属性进行设置。例如：

```
QLineEdit {
    background-image: url(:/images/yellow-bg.png);
    background-position: top right;
    background-repeat: repeat-y;
}
```

如果指定了背景图片和背景颜色，背景颜色将会在图片的半透明区域中透射出来。

目前为止，使用的所有的选择器都是类的名字。还有其他的一些可以使用的选择器，它们列在图 19.2 中。例如，如果想给 OK 和 Cancel 按钮指定前景颜色，可以这样写：

```
QPushButton[text="OK"] {
    color: green;
}

QPushButton[text="Cancel"] {
    color: red;
}
```

选择器	实例	可以匹配的窗口部件
全局对象	*	任意窗口部件
类型	QDial	给定类的实例，包括子类
类	.QDial	给定类的实例，不包括子类
标识	QDial # ageDial	给定对象名称的窗口部件
Qt 属性	QDial[y = "0"]	为某些属性赋值的窗口部件
子对象	QFrame > QDial	给定窗口部件的直接子窗口部件
子孙对象	QFrame QDial	给定窗口部件的子孙窗口部件

图 19.2　样式表选择器

这种选择器语法对任何的 Qt 属性都适用。尽管如此，必须记住，样式表不会注意到属性值的修改。选择器能以各种方式组合。例如，为了选择所有称作“okButton”的 QPushButton，它们的 x 和 y 属性为 0，名为“frame”的 QFrame 直接子对象，可以这样写：

```
QFrame#frame > QPushButton[x="0"][y="0"]#okButton {
    ...
}
```

在一个拥有大量窗体和编辑框、组合框的应用程序中，例如那些在各种机构中使用的，通常必选字段会使用 yellow 作为背景色。我们假设将在应用程序中提供这一通用功能。首先，应该使用这个样式表：

```
*[mandatoryField="true"] {
    background-color: yellow;
}
```

尽管在 Qt 中没有 mandatoryField 属性的定义，但很容易通过调用 QObject::setProperty()创建一个。从 Qt 4.2 开始，动态地设置一个不存在的属性的值可以创建这一属性。例如：

```
nameLineEdit->setProperty("mandatoryField", true);
genderComboBox->setProperty("mandatoryField", true);
ageSpinBox->setProperty("mandatoryField", true);
```

样式表不仅仅在控制颜色时有用,它们也可以用来对窗口部件元素的大小和位置进行调整。例如,下面的规则可以增加复选框和单选钮的状态指示器的大小为 20×20 像素,并且确保指示器和说明文字之间有 8 像素的间隙:

```
QCheckBox::indicator, QRadioButton::indicator {
    width: 20px;
    height: 20px;
}

QCheckBox, QRadioButton {
    spacing: 8px;
}
```

注意第一个规则的选择器语法。如果写的是 QCheckBox,而不是 QCheckBox::indicator,就指定了整个窗口部件的尺寸,而非指示器了。第一个规则如图 19.3 所示。

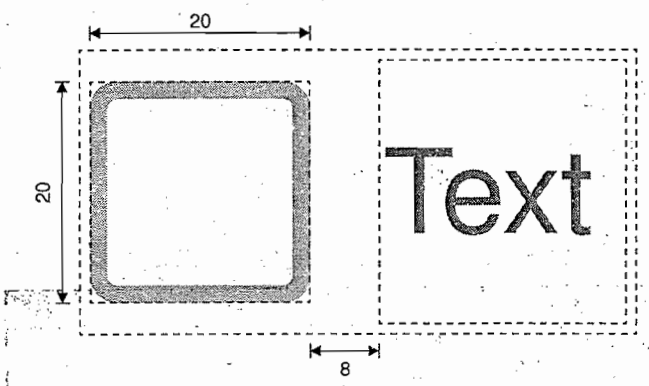


图 19.3 设置一个 QCheckBox 指示器的大小

一些辅助控制器(subcontrol),如::indicator,可以跟一些窗口部件一样使用。图 19.4 列出了 Qt 所支持的一些辅助控制器。

辅助控制器	说明
::indicator	复选框、单选钮、可选菜单项或可选群组框的指示器
::menu-indicator	按钮的菜单指示器
::item	菜单、菜单栏或状态栏项
::up-button	微调框或滚动条的向上按钮
::down-button	微调框或滚动条的向下按钮
::up-arrow	微调框、滚动条或标题视图的向上箭头
::down-arrow	微调框、滚动条、标题视图或组合框的向下箭头
::drop-down	组合框的下拉箭头
::title	群组框的标题

图 19.4 最常见的自定义辅助控制器

除了辅助控制器,样式表还可以用来指定窗口部件的各个状态。例如,当鼠标在复选框的文本上悬停时,我们可能想用白色指定它的:hover 状态:

```
QCheckBox:hover {
    color: white;
}
```

状态由单个的冒号决定,而辅助控制器由两个冒号指定。我们可以一个接一个地列出几个状

态，它们彼此用冒号隔开。在这种情况下，当窗口部件满足所有的状态时，规则才被使用。例如，下面的规则只有当鼠标在一个被选中的复选框上悬停时规则才被使用：

```
QCheckBox:checked:hover {
    color: white;
}
```

如果希望在任何一个状态为 true 的情况下使用规则，则可以使用多个选择器，用逗号把它们隔开：

```
QCheckBox:hover, QCheckBox:checked {
    color: white;
}
```

逻辑否可以用感叹号(!)表示：

```
QCheckBox:!checked {
    color: blue;
}
```

状态可以与辅助控制器合用：

```
QComboBox::drop-down:hover {
    image: url(:/images/downarrow_bright.png);
}
```

图 19.5 列出了可用的样式表状态。

状态	说明
:disabled	禁用窗口部件
:enabled	启用窗口部件
:focus	窗口部件有输入焦点
:hover	鼠标在窗口部件上悬停
:pressed	鼠标按键点击窗口部件
:checked	按钮已被选中
:unchecked	按钮未被选中
:indeterminate	按钮被部分选中
:open	窗口部件位于打开或扩展状态
:closed	窗口部件位于关闭或销毁状态
:on	窗口部件的状态是“on”
:off	窗口部件的状态是“off”

图 19.5　窗口部件中一些可以访问样式表的状态

样式表也可以与其他的技术结合实现更复杂的自定义效果。例如，假设我们想在 QLineEdit 窗体中安放一个微型“erase”按钮，在 QLineEdit 文字的右边。这包括创建一个 EraseButton 类，把它放在 QLineEdit 的上方(例如使用布局管理器)，但也给按钮留一些空间，这样能够避免输入的文字与按钮的冲突。通过子类化 QStyle 实现这一功能是不方便的，因为我们可能需要子类化 Qt 中任何被此应用程序使用的样式(QWindowsVistaStyle、QPlastiqueStyle 等)。使用样式表，下面的规则是一个窍门：

```
QLineEdit {
    padding: 0px 15px 0px 0px;
}
```

padding 属性允许我们指定窗口部件的上边、右边、下边和左边的填充空间。填充的位置在 QLineEdit 的文字和窗体之间。为了更加方便，CSS 同时定义了 padding-top、padding-right、padding-

bottom 和 padding-left,它们适用于只想设置一个填充值的情况。例如:

```
QLineEdit {
    padding-right: 15px;
}
```

如同大多数的 Qt 窗口部件都可以使用样式表定制一样,QLineEdit 可以支持如图 19.6 所示的盒子模型。此模型可指定 4 个影响布局的矩形,从而绘制一个自定义的窗口部件:

1. contents rectangle 是最里面的矩形。它是绘制窗口部件内容(例如文字或者图片)的地方。
2. padding rectangle 包围 contents rectangle。它负责由 padding 属性指定填充操作。
3. border rectangle 包围 padding rectangle。它为边界预留空间。
4. margin rectangle 是最外边的矩形。它包围 border rectangle,负责任何指定的边缘空白区域。

对于没有 padding、border 和 margin 的普通窗口部件,这 4 个矩形重合在一起。

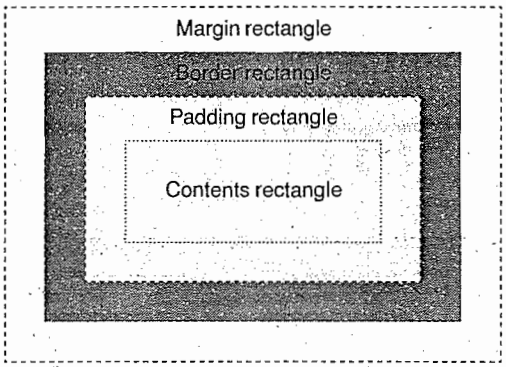


图 19.6 CSS 盒子模型

现在,介绍一个实现自定义的称作 Candy 的样式表。图 19.7 显示了一个 Candy 样式窗口部件的选择器。Candy 样式使用如图 19.6 所示的盒子模型为 QLineEdit、QListView、QPushButton 和 QComboBox 实现自定义的外观。下面将逐步介绍这一样式表,全部的样式表 qss/candy.qss 可以在 Candy 例子的目录下找到,与本书的例子放在一起。

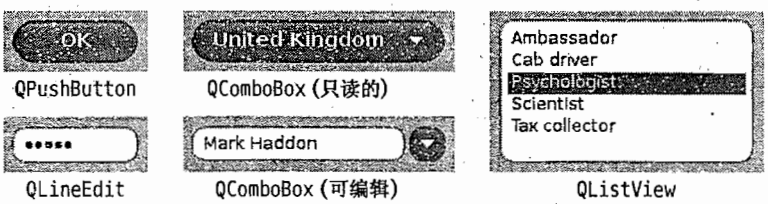


图 19.7 Candy 风格的窗口部件

对话框中的窗口部件如图 19.1 所示。对话框的背景图片使用以下的规则设置:

```
QDialog {
    background-image: url(:/images/background.png);
}
```
[R1]

下面的规则集为这些 QLabel 设置了 color 和 font 属性:

```
QLabel {
    font: 9pt;
    color: rgb(0, 0, 127);
}
```
[R2]

随后的规则集定义了对话框中的 QLineEdit 和 QListView 的外观：

```
QLineEdit,
QListView {
    color: rgb(127, 0, 63);
    background-color: rgb(255, 255, 241);
    selection-color: white;
    selection-background-color: rgb(191, 31, 127);
    border: 2px groove gray;
    border-radius: 10px;
    padding: 2px 4px;
}
```

[R3]

为了使 QLineEdit 和 QListView 与众不同，我们需要为普通文本和选中的文本指定前景色和背景色。此外，我们使用 border 属性指定了一个灰色、2 像素宽的“凹槽”边框，可以分别设置为 border-width，border-style 以及 border-color，由其代替 border；通过指定 border-radius，可以把边框的角设为圆角，这里的半径为 10 像素。图 19.8 提供了一个图示，它是修改窗口部件 border 和 padding 属性的效果图。为了保证窗口部件内容与边框的圆角不重合，我们指定了一个竖直方向 2 像素、水平方向 4 像素的内部填充区域。对于 QListView，竖直方向的填充看起来不正确，因此可以用以下的方法覆盖它：

```
QListView {
    padding: 5px 4px;
}
```

[R4]

当一个属性被具有同一选择器的几个规则同时设置时，那么只有最后一个规则将起作用。

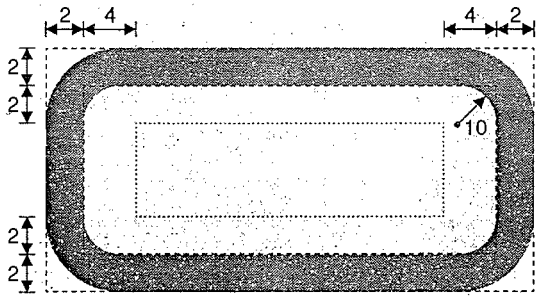
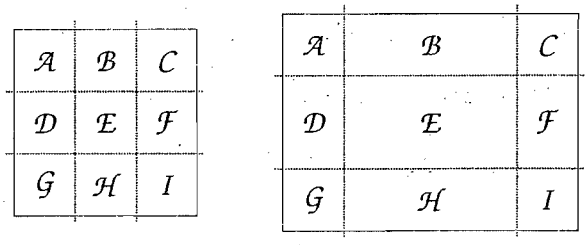


图 19.8　QLineEdit 的结构

对于 QPushButton 的定制，我们将使用一种完全不同的方式。用一个准备好的图像作为背景，代替使用样式表规则绘制按钮的方法。同样，为了使按钮可以缩放，按钮的背景使用 CSS 的边界图（border image）机制定义。

与使用 background-image 定义的背景图像不同，边界图被分隔成 3 × 3 的小格，如图 19.9 所示。当填充窗口部件背景时，4 个角（图中的格子 $\mathcal{A}$、$\mathcal{C}$、$\mathcal{G}$ 以及 $\mathcal{I}$）保持不变，其他 5 个格子被拉伸或者平铺，填充可用空间。



(a) 原始图像　　(b) 结果图像

图 19.9　边界图的划分

使用 border-image 属性可以指定各个边界图,它要求指定一个图像文件名和定义 9 个格子的 4 条“切线”。切线用其到上、右、下和左边缘的距离定义。border.png 作为边界图,距离上、右、下和左边缘的切线为 4、8、12 和 16,应该这样定义:

```
border-image: url(border.png) 4 8 12 16;
```

当使用边界图时,必须显式地设置 border-width 属性。一般情况下,border-width 应该与切线的位置一致;否则,为了与 border-width 相符合,角上的格子将被拉伸或缩短。对于 border.png 的例子,应该这样指定边框宽度:

```
border-width: 4px 8px 12px 16px;
```

既然已经知道了边界图的工作机制,我们就可以看一下如何使用它来绘制 Candy 风格的 QPushButton。下面就是定义在正常情况下绘制按钮的那些规则:

```
QPushButton {
    color: white;
    font: bold 10pt;
    border-image: url(:/images/button.png) 16;
    border-width: 16px;                                    [R5]
    padding: -16px 0px;
    min-height: 32px;
    min-width: 60px;
}
```

在 Candy 样式表中,QPushButton 边界图的 4 条切线位于距离 34 × 34 的边界图 16 像素的位置,如图 19.10(a)所示。因为 4 条切线是统一的,只需要用“16”定义切线,“16px”定义边框宽度。

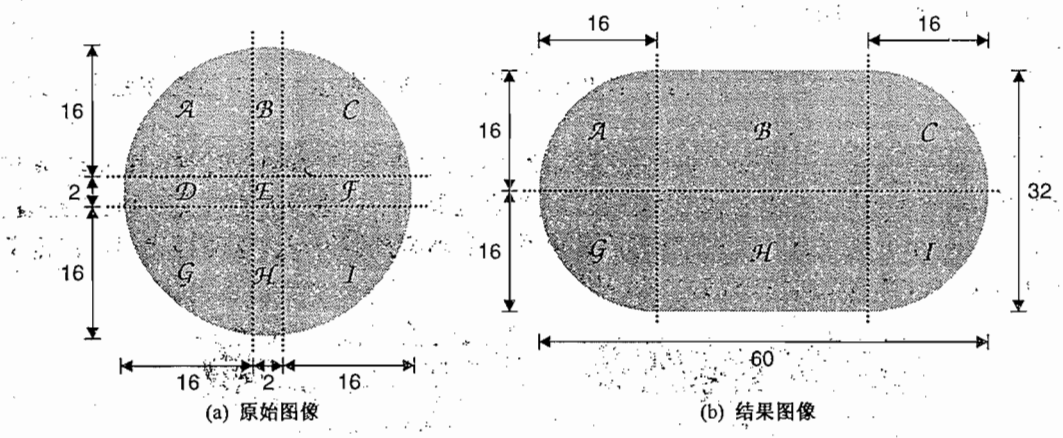


图 19.10 用于 QPushButton 的边界图

在图 19.10(b)所示的 QPushButton 例子中,对应于 $\mathcal{D}$、$\mathcal{E}$、和 $\mathcal{F}$ 的边界图的格子被丢弃了,因为缩小的按钮不足以显示,会水平拉伸格子 $\mathcal{B}$ 和 $\mathcal{H}$,以便可以占用余下的宽度。

边界图的标准用途是在窗口部件周围提供一个边框,使得窗口部件在边框内。但我们推翻了边界图的这种机制,用它来创建窗口部件的背景。结果,格子 $\mathcal{E}$ 被丢弃了,填充矩形的高度变成了 0。为了给按钮的文字留些空间,我们指定一个 -16 像素的竖直填充空间。图 19.11 显示了这种情况。如果用边界图的机制定义一个实际的边框,我们可能不希望让它与文字冲突——因为我们使用它来创建一个可缩放的背景,想把文字放在它的上面而不是内部。

可以使用 min-width 和 min-height 设置按钮尺寸的最小值。这里选定的值确保有足够的空间留给角上的边界图,并且确保 OK 按钮比需要的稍大一些,使得与 Cancel 按钮相邻的效果看起来更好一些。

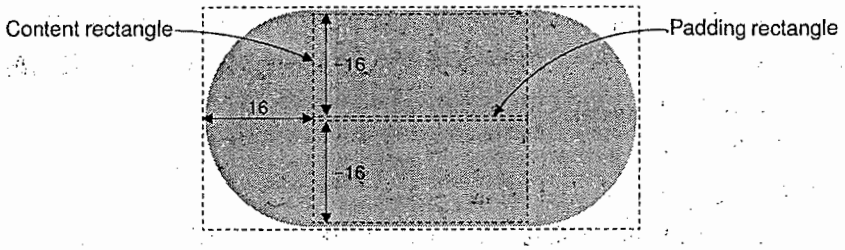


图 19.11　QPushButton 的结构

前面的 QPushButton 规则应用于所有的按钮。现在，我们将定义一些额外的规则，这些规则只在按钮处于特定的状态时才会用到。

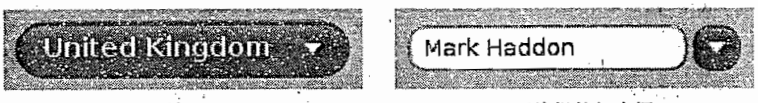
```
QPushButton:hover {
    border-image: url(:/images/button-hover.png) 16;
}
```
[R6]

当鼠标的指针位于 QPushButton 的上方时，:hover 的状态变成了 true，指定的规则会覆盖具有较少规则选择器的其他规则。这里，我们使用这一技术指定一个更明亮的图像作为边界图，使其具有更好的悬停效果。前面指定的其他的 QPushButton 的属性仍然适用，只有 border-image 的属性改变。

```
QPushButton:pressed {
    color: lightgray;
    border-image: url(:/images/button-pressed.png) 16;
    padding-top: -15px;
    padding-bottom: -17px;
}
```
[R7]

当用户单击按钮时，可以把前景色改为浅灰色，使用一个稍暗的边界图，并且把按钮中的文字下移一个像素，对填充区域做小的调整。

最后的风格规则是自定义 QComboBox 的外观。为了达到控制和细致的目的，使用样式表，将区分只读和可编辑的组合框，如图 19.12 所示。只读的组合框是一个右边带下拉箭头的 QPushButton，可编辑的组合框则由一个类似 QLineEdit 的组件以及一个小圆按钮组成。这使得我们可以定义 QLineEdit、QListView 和 QPushButton 的大多数规则。



(a) 只读的组合框　　(b) 可编辑的组合框

图 19.12　具有 Candy 风格的 QComboBox

- 定义 QLineEdit 的规则可以用来定义可编辑的组合框：

```
QComboBox:editable,
QLineEdit,
QListView {
    color: rgb(127, 0, 63);
    background-color: rgb(255, 255, 241);
    selection-color: white;
    selection-background-color: rgb(191, 31, 127);
    border: 2px groove gray;
    border-radius: 10px;
    padding: 2px 4px;
}
```
[R3′]

- 定义 QPushButton 普通状态的规则可以扩展到定义只读组合框：

```
QComboBox:!editable,
QPushButton {
    color: white;
    font: bold 10pt;
    border-image: url(:/images/button.png) 16;
    border-width: 16px;
    padding: -16px 0px;
    min-height: 32px;
    min-width: 60px;
}
```
[R5′]

- 鼠标在只读组合框或者下拉按钮上悬停的规则可以修改背景图片,如在 QPushButton 上应用的规则:

```
QComboBox:!editable:hover,
QComboBox::drop-down:editable:hover,
QPushButton:hover {
    border-image: url(:/images/button-hover.png) 16;
}
```
[R6′]

- 点击一个只读的按钮如同点击一个 QPushButton:

```
QComboBox:!editable:on,
QPushButton:pressed {
    color: lightgray;
    border-image: url(:/images/button-pressed.png) 16;
    padding-top: -15px;
    padding-bottom: -17px;
}
```
[R7′]

重用规则 R3、R5、R6 和 R7 可以为我们节省不少时间,也有助于保持风格的统一。但我们还没有定义下拉按钮的规则,所以现在就来创建它:

```
QComboBox::down-arrow {
    image: url(:/images/down-arrow.png);
}
```
[R8]

我们提供了自己的下拉箭头的图像,以便它可以比标准的箭头稍大一些:

```
QComboBox::down-arrow:on {
    top: 1px;
}
```
[R9]

如果组合框是打开的,向下箭头将下移一个像素:

```
QComboBox * {
    font: 9pt;
}
```
[R10]

当用户单击了组合框,它显示一个项的列表。规则 R10 确保了组合框弹出的对象(或者任何子窗口部件)没有继承规则 R5′中指定的大字体:

```
QComboBox::drop-down:!editable {
    subcontrol-origin: padding;
    subcontrol-position: center right;
    width: 11px;
    height: 6px;
    background: none;
}
```
[R11]

使用 subcontrol-origin 和 subcontrol-position 属性,下拉箭头竖直居中,位于只读组合框的填充矩形的右侧。此外,还设置它的大小与按钮内容(11 × 6 像素的 down-arrow.png 图像)的大小相符,我们把背景设置为无,因为下拉按钮仅由下拉箭头组成。

```
QComboBox:!editable {
    padding-right: 15px;
}
```
[R12]

对于只读组合框，我们指定一个右置的 15 像素的填充区域，确保组合框的显示文字与下拉箭头不重合。图 19.13 显示了这些元素彼此的位置关系。

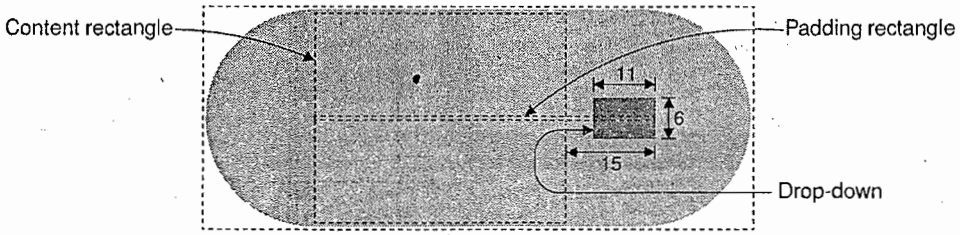


图 19.13　一个只读 QComboBox 的结构

对于可编辑组合框，需要配置下拉按钮，使它看起来像一个微型按钮：

```
QComboBox::drop-down:editable {
    border-image: url(:/images/button.png) 16;
    border-width: 10px;
    subcontrol-origin: margin;
    subcontrol-position: center right;
    width: 7px;
    height: 6px;
}
```
[R13]

我们指定 button.png 作为边界图。然而，这次指定一个宽 10 像素的边框代替 16 像素，并按比例缩小图像，指定一个水平 7 像素竖直 6 像素的固定内容尺寸。图 19.14 是这种做法的示意图。

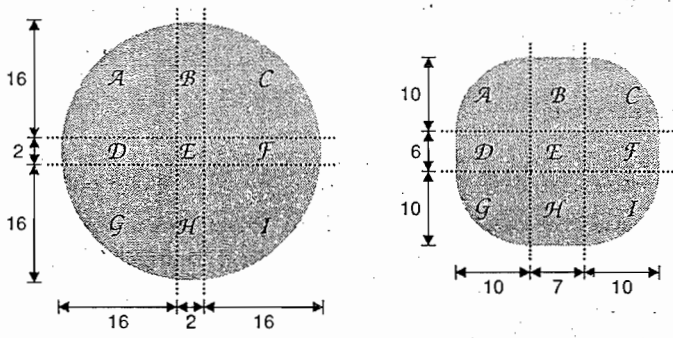


图 19.14　可编辑型 QComboBox 下拉按钮的边界图

如果组合框打开，就使用一个不同的、暗色的图像作为下拉按钮：

```
QComboBox::drop-down:editable:open {
    border-image: url(:/images/button-pressed.png) 16;
}
```
[R14]

对于可编辑组合框，指定一个右置的 29 像素的区域为下拉按钮提供空间，如图 19.15 所示：

```
QComboBox:editable {
    margin-right: 29px;
}
```
[R15]

现在就完成了 Candy 样式表的创建。样式表的长度大约有 100 行，并且使用了几个自定义的图像。这是个与众不同的对话框。

使用样式表创建自定义样式需要不断摸索，特别是对那些以前没有用过 CSS 的人。使用样式表的一个主要挑战是 CSS 的冲突方案和层叠的处理比较抽象。更详细的信息请参考在线帮助文档 http://doc.trolltech.com/4.3/stylesheet.html，它描述了 Qt 的样式表支持，提供 CSS 规范的链接。

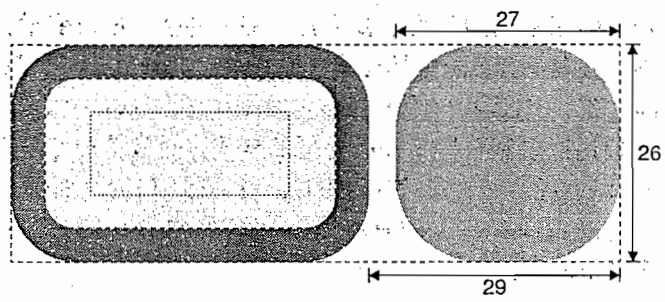


图 19.15 一个可编辑的 QComboBox 的结构

19.2 子类化 QStyle

QStyle 类是在 Qt 2.0 中引入的,它提供了一种包装应用程序外观的方法。例如 QWindowsStyle、QMotifStyle 和 QCDEStyle,这些类为 Qt 运行的平台和桌面系统实现了外观。Qt 4.3 提供 8 种样式,还有 QStyle 的抽象基类和便利的 QCommonStyle 基类。图 19.16 显示了它们之间的关系。

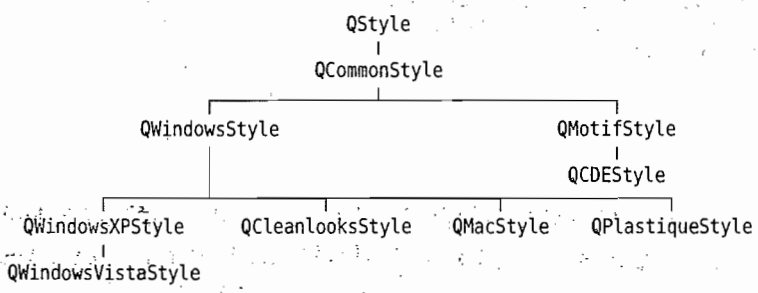


图 19.16 Qt 内置样式的继承树

对于 Qt 开发者来说,QStyle 体系能够通过子类化 QStyle 或者以现存的样式来开发新的自定义外观。在现存的样式上(例如 QWindowStyle),我们可以做最少的修改,或者可以从无到有地创建一种完全自定义的样式。

QStyle API 包含绘制图形元素[drawPrimitive()、drawControl()、drawComplexControl()等]的函数,以及样式查询函数[pixelMetrics()、styleHint()、hitTest()等]。QStyle 成员函数典型地带有 QStyleOption 对象,它包含绘制窗口部件的通用信息(例如其调色板)以及特有信息(例如按钮的文字)。函数还包含一个可选的 QWidget 指针,以应对 QStyleOption 不能提供全部所需信息的情况。

假设我们想创建一个 MyPushButton 类,它类似于标准的 Qt 按钮,但不从 QPushButton 继承。(这个例子有些牵强,但它将帮助我们弄清窗口部件和样式之间的关系。)在 MyPushButton 绘图事件处理器中,我们将创建一个 QStyleOption(实际上是 QStyleOptionButton),并且会调用 QStyle::drawControl():

```
void MyPushButton::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);

    QStyleOptionButton option;
    option.initFrom(this);
    if (isFlat())
        option.features |= QStyleOptionButton::Flat;
    option.text = text();

    style()->drawControl(QStyle::CE_PushButton, &option, &painter,
                         this);
}
```

QStyleOption::initFrom()函数初始化用于显示窗口部件的基本成员变量，例如 rect、state(是否可用、是否获得焦点，等等)，以及调色板。具有特性的 QStyleOptionButton 的成员变量必须被手动地初始化。在 MyPushButton 例子中，我们初始化 features 和 text，允许 icon 和 iconSize 使用默认值。

QWidget::style()函数返回绘制窗口部件的合适的样式。通常使用 QApplication::setStyle()设置整个应用程序的样式，但也可以覆盖它，使用 QWidget::setStyle()为个别窗口部件设置样式。

drawControl()函数被各种 QStyle 的子类重新实现，用于绘制 QPushButton 和其他的一些简单的窗口部件。典型的实现如下所示：

```
void QMotifStyle::drawControl(ControlElement element,
                              const QStyleOption *option,
                              QPainter *painter,
                              const QWidget *widget) const
{
    switch (element) {
    case CE_CheckBox:
        ...
    case CE_RadioButton:
        ...
    case CE_PushButton:
        if (const QStyleOptionButton *buttonOption =
                qstyleoption_cast<const QStyleOptionButton *>(option)) {
            // draw push button
        }
        break;
        ...
    }
}
```

第一个参数(element)指出将要绘制的窗口部件的类型。如果类型是 CE_PushButton，option 参数将被强制转换为 QStyleOptionButton。如果强制转换成功，将绘制 QStyleOptionButton 描绘的按钮。从 const QStyleOption * 向 const QStyleOptionButton * 的强制转换使用 qstyleoption_cast <T>() 来完成。如果 option 指向的不是 QStyleOptionButton 实例，该强制转换将返回一个空指针。

除了使用 QStyleOption，QStyle 的子类也可以直接查询窗口部件：

```
case CE_PushButton:
    if (const QPushButton *button =
            qobject_cast<const QPushButton *>(widget)) {
        // draw push button
    }
    break;
```

这个版本的缺点是样式的代码依赖于 QPushButton，因此不能用于绘制，比如，MyPushButton。因为这个原因，当内置的 QStyle 的子类需要获取绘制窗口部件的参数信息时使用 QStyleOption 参数，只能从 QWidget 获取的参数就从 QWidget 获取。

在本节余下的部分，将介绍如图 19.17 所示的 Bronze 样式的代码。Bronze 样式实现了带渐变背景的圆角按钮、非传统的微调框、夸张的选取标志和“棕色画刷”背景。为了实现这一点，我们应用了高级二维图形特性，例如反走样、半透明、线性渐变以及组合模式。

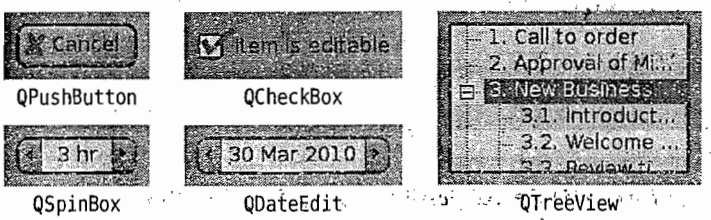


图 19.17　具有 Bronze 风格的一些窗口部件

以下是 BronzeStyle 类的定义:

```
class BronzeStyle : public QWindowsStyle
{
    Q_OBJECT

public:
    BronzeStyle() {}

    void polish(QPalette &palette);
    void polish(QWidget *widget);
    void unpolish(QWidget *widget);
    int styleHint(StyleHint which, const QStyleOption *option,
                  const QWidget *widget = 0,
                  QStyleHintReturn *returnData = 0) const;
    int pixelMetric(PixelMetric which, const QStyleOption *option,
                    const QWidget *widget = 0) const;
    void drawPrimitive(PrimitiveElement which,
                       const QStyleOption *option, QPainter *painter,
                       const QWidget *widget = 0) const;
    void drawComplexControl(ComplexControl which,
                            const QStyleOptionComplex *option,
                            QPainter *painter,
                            const QWidget *widget = 0) const;
    QRect subControlRect(ComplexControl whichControl,
                         const QStyleOptionComplex *option,
                         SubControl whichSubControl,
                         const QWidget *widget = 0) const;

public slots:
    QIcon standardIconImplementation(StandardPixmap which,
                                     const QStyleOption *option,
                                     const QWidget *widget = 0) const;

private:
    void drawBronzeFrame(const QStyleOption *option,
                         QPainter *painter) const;
    void drawBronzeBevel(const QStyleOption *option,
                         QPainter *painter) const;
    void drawBronzeCheckBoxIndicator(const QStyleOption *option,
                                     QPainter *painter) const;
    void drawBronzeSpinBoxButton(SubControl which,
                                 const QStyleOptionComplex *option,
                                 QPainter *painter) const;
};
```

当创建一个自定义样式时,我们通常会基于某种存在的样式,这样就不用从零开始做所有的事情。对于这个例子,我们选择 QWindowsStyle,这是一个经典的 Windows 样式。尽管 Bronze 样式不是很像 Windows 样式,但在 QWindowsStyle 和它的基类 QCommonStyle 中,大部分代码都可以重用,以实现所想到的任何样式。这就是 QMacStyle 从 QWindowsStyle 派生的原因,尽管它们看起来很不同。

BronzeStyle 实现了由 QStyle 定义的几个公有函数。polish()和 unpolish()函数会在安装或者卸载样式的时候得到调用。它们可以让我们对窗口部件或调色板做适当修改。其他的公有函数有的是查询函数[styleHint(),pixelMetric(),subControlRect()],有的是用于绘图的函数[drawPrimitive(),drawComplexControl()]。

BronzeStyle 也提供了一个称为 standardIconImplementation()的公有槽。这个槽会被 Qt 的内省(introspection)机制发现,尽管它是一个虚函数,但在必要时也可调用。Qt 有时利用这一惯例来添加虚函数以保证与以前的 Qt 4 版本的二进制兼容性。在 Qt 5 中有望用 standardIcon()虚函数代替 standardIconImplementation()槽。

BronzeStyle 类也定义了几个私有成员函数。在介绍完公有函数之后再介绍它们。

```
void BronzeStyle::polish(QPalette &palette)
{
    QPixmap backgroundImage(":/images/background.png");
```

```
    QColor bronze(207, 155, 95);
    QColor veryLightBlue(239, 239, 247);
    QColor lightBlue(223, 223, 239);
    QColor darkBlue(95, 95, 191);

    palette = QPalette(bronze);
    palette.setBrush(QPalette::Window, backgroundImage);
    palette.setBrush(QPalette::BrightText, Qt::white);
    palette.setBrush(QPalette::Base, veryLightBlue);
    palette.setBrush(QPalette::AlternateBase, lightBlue);
    palette.setBrush(QPalette::Highlight, darkBlue);
    palette.setBrush(QPalette::Disabled, QPalette::Highlight,
                     Qt::darkGray);
}
```

Bronze 样式的一个突出的特点是它的颜色配置。无论用户在窗口系统中设置了怎样的颜色，Bronze 样式都有一个棕色的外观。自定义外观的颜色配置有两种设置方式：忽略窗口部件的 QPalette，使用我们喜欢的颜色（棕色、亮棕色、暗棕色等）进行绘制，或者重新实现 QStyle::polish (QPalette &)调整应用程序或者窗口部件的调色板，然后使用该调色板。第二种方法更灵活，因为可以在子类中（比如，SilverStyle）通过重新实现 polish()覆盖原有的配色方案。

抛光（polishing）的概念在窗口部件中是通用的。当样式应用到窗口部件时，polish(QWidget *) 就会得以调用，从而允许我们执行最后的定制：

```
void BronzeStyle::polish(QWidget *widget)
{
    if (qobject_cast<QAbstractButton *>(widget)
            || qobject_cast<QAbstractSpinBox *>(widget))
        widget->setAttribute(Qt::WA_Hover, true);
}
```

这里，我们重新实现了 polish(QWidget *)，以便在按钮和微调框中设置 Qt::WA_Hover 属性。当这一属性被设置、鼠标进入或者离开窗口部件所在的区域时，会产生一个绘制事件。这给我们一个机会，可以根据鼠标是否位于窗口部件上把窗口部件绘制出不同的样子。

在窗口部件创建完之后并且在第一次显示之前，会使用当前的样式来调用这个函数，然后，只有在当前样式动态改变时被调用。

```
void BronzeStyle::unpolish(QWidget *widget)
{
    if (qobject_cast<QAbstractButton *>(widget)
            || qobject_cast<QAbstractSpinBox *>(widget))
        widget->setAttribute(Qt::WA_Hover, false);
}
```

正如在窗口部件上应用样式时会调用 polish(QWidget *)一样，当动态改变样式时，同样会调用 unpolish(QWidget *)。unpolish()的目的是取消 polish()的作用，这样窗口部件就可以被一个新的样式抛光。优秀的样式能够取消它们在 polish()函数中的操作。

polish(QWidget *)的一般用法是把样式子类作为窗口部件上的事件过滤器。这对于一些更高级的定制很有用。例如，QWindowsVistaStyle 和 QMacStyle 使用这种技术使默认按钮具有动画效果。

```
int BronzeStyle::styleHint(StyleHint which, const QStyleOption *option,
                           const QWidget *widget,
                           QStyleHintReturn *returnData) const
{
    switch (which) {
    case SH_DialogButtonLayout:
        return int(QDialogButtonBox::MacLayout);
    case SH_EtchDisabledText:
        return int(true);
    case SH_DialogButtonBox_ButtonsHaveIcons:
        return int(true);
```

```
    case SH_UnderlineShortcut:
        return int(false);
    default:
        return QWindowsStyle::styleHint(which, option, widget,
                                        returnData);
    }
}
```

styleHint()函数返回一些关于样式外观的提示。例如,对于SH_DialogButtonLayout返回MacLayout,表示我们想要QDialogButtonBox依从Mac OS X的风格,OK在Cancel的右边。styleHint()的返回类型是int。对于不能用整数表示的少数的样式提示,styleHint()提供了一个指向QStyleHintReturn对象的指针。

```
int BronzeStyle::pixelMetric(PixelMetric which,
                             const QStyleOption *option,
                             const QWidget *widget) const
{
    switch (which) {
    case PM_ButtonDefaultIndicator:
        return 0;
    case PM_IndicatorWidth:
    case PM_IndicatorHeight:
        return 16;
    case PM_CheckBoxLabelSpacing:
        return 8;
    case PM_DefaultFrameWidth:
        return 2;
    default:
        return QWindowsStyle::pixelMetric(which, option, widget);
    }
}
```

pixelMetric()函数返回一个像素值,用于用户界面元素中。通过重新实现这一功能,既影响内置Qt窗口部件的绘制,也影响其尺寸。

对于PM_ButtonDefaultIndicator返回0,因为不希望在默认的按钮旁边保留额外的空间(QWindowsStyle默认的是1像素)。对于复选框,PM_IndicatorWidth和PM_IndicatorHeight控制指示器的大小(通常是一个小正方形),而PM_CheckBoxLabelSpacing控制选择指示器与其右边的文字之间的距离(如图19.18所示)。最后,PM_DefaultFrameWidth定义QFrame、QPushButton、QSpinBox以及其他的一些窗口部件的线宽。对于其他的PM_*xxx*值,我们从基类中继承像素规格的值。

```
QIcon BronzeStyle::standardIconImplementation(StandardPixmap which,
        const QStyleOption *option, const QWidget *widget) const
{
    QImage image = QWindowsStyle::standardPixmap(which, option, widget)
                   .toImage();
    if (image.isNull())
        return QIcon();
    QPalette palette;
    if (option) {
        palette = option->palette;
    } else if (widget) {
        palette = widget->palette();
    }

    QPainter painter(&image);
    painter.setOpacity(0.25);
    painter.setCompositionMode(QPainter::CompositionMode_SourceAtop);
    painter.fillRect(image.rect(), palette.highlight());
    painter.end();

    return QIcon(QPixmap::fromImage(image));
}
```

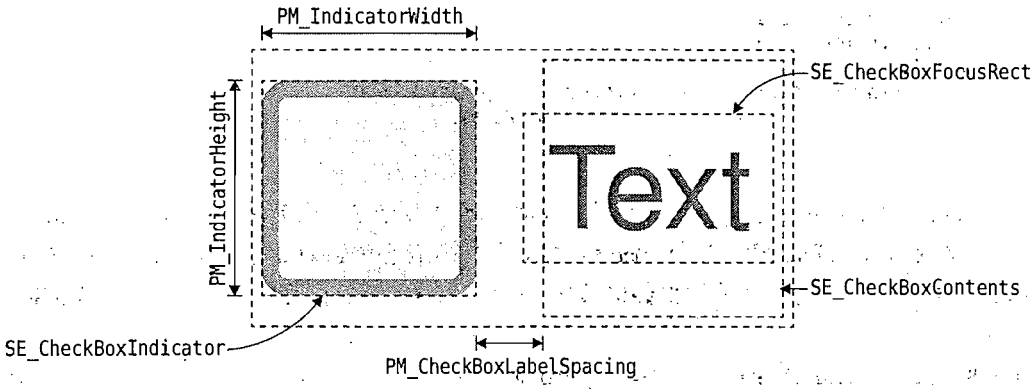


图 19.18 QCheckBox 的结构

如同前面介绍的那样，Qt 调用 standardIconImplementation()槽获取用在用户界面上的标准图标。我们调用基类的 standardPixmap()获取图标，并绘制浅蓝色，使其与余下的风格相协调。着色可以通过在图标上绘制 25%不透明的蓝色实现。通过使用 SourceAtop 复合模式，我们确保了原来的透明区域依然是透明区域，而不是变成了 25%的蓝色和 75%的透明色。第 8 章中“用 QImage 高质量绘图”一节(见 8.3 节)中介绍过复合模式。

```
void BronzeStyle::drawPrimitive(PrimitiveElement which,
                                const QStyleOption *option,
                                QPainter *painter,
                                const QWidget *widget) const
{
    switch (which) {
    case PE_IndicatorCheckBox:
        drawBronzeCheckBoxIndicator(option, painter);
        break;
    case PE_PanelButtonCommand:
        drawBronzeBevel(option, painter);
        break;
    case PE_Frame:
        drawBronzeFrame(option, painter);
        break;
    case PE_FrameDefaultButton:
        break;
    default:
        QWindowsStyle::drawPrimitive(which, option, painter, widget);
    }
}
```

Qt 调用 drawPrimitive()函数绘制“基本的”用户界面元素，这些元素会被几个窗口部件使用。例如，PE_IndicatorCheckBox 会被 QCheckBox、QGroupBox 和 QItemDelegate 用来绘制选择指示器。

Bronze 样式重新实现了 drawPrimitive()，给选择指示器、按钮和边框定制一种外观。图 19.19 所示就是 QPushButton 的结构，它是 Bronze 样式必须处理的。drawBronzeCheckBoxIndicator()、drawBronzeBevel()和 drawBronzeFrame()函数都是私有函数，稍后将会介绍它们。

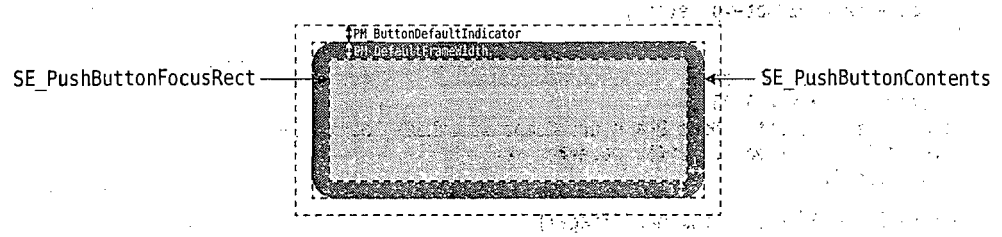


图 19.19 QPushButton 的结构

对于PE_FrameDefaultButton,我们什么也不用做,因为我们不想在默认的按钮旁另外再绘制一个多余的边框。对于其他的基本元素,只是简单地调用基类的函数。

```
void BronzeStyle::drawComplexControl(ComplexControl which,
                                     const QStyleOptionComplex *option,
                                     QPainter *painter,
                                     const QWidget *widget) const
{
    if (which == CC_SpinBox) {
        drawBronzeSpinBoxButton(SC_SpinBoxDown, option, painter);
        drawBronzeSpinBoxButton(SC_SpinBoxUp, option, painter);

        QRect rect = subControlRect(CC_SpinBox, option,
                                    SC_SpinBoxEditField)
                     .adjusted(-1, 0, +1, 0);
        painter->setPen(QPen(option->palette.mid(), 1.0));
        painter->drawLine(rect.topLeft(), rect.bottomLeft());
        painter->drawLine(rect.topRight(), rect.bottomRight());
    } else {
        return QWindowsStyle::drawComplexControl(which, option, painter,
                                                 widget);
    }
}
```

Qt调用drawComplexControl()函数绘制多重辅助控制器窗口部件——特别是QSpinBox。因为想给QSpinBox一个全新的外观,所以重新实现了drawComplexControl(),处理CC_SpinBox的情况。

要想绘制QSpinBox,必须绘制向上和向下的按钮,以及整个微调框周围的边框。(QSpinBox的结构如图19.20所示。)因为用来绘制向上按钮的代码几乎与绘制向下按钮的代码相同,我们把它提出来写到drawBronzeSpinBoxButton()私有函数中。该函数也绘制整个微调框周围的边框。

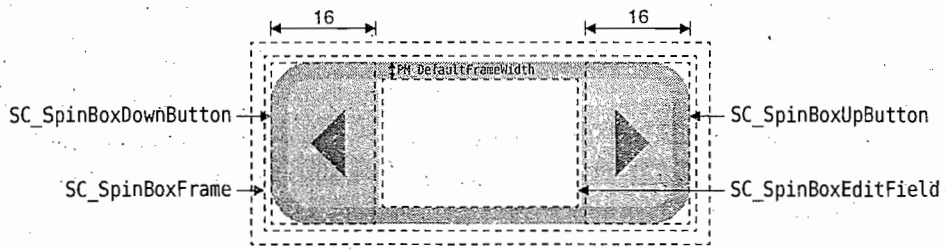


图19.20 QSpinBox的结构

QSpinBox使用QLineEdit显示可编辑的部分,因此不需要绘制这一部分。尽管如此,为了清楚地区分开QLineEdit和微调框按钮,需要在QLineEdit的边缘绘制两条亮棕色的竖线。QLineEdit的空间大小可以通过调用subControlRect()获得,SC_SpinBoxEditField作为第三个参数。

```
QRect BronzeStyle::subControlRect(ComplexControl whichControl,
                                  const QStyleOptionComplex *option,
                                  SubControl whichSubControl,
                                  const QWidget *widget) const
{
    if (whichControl == CC_SpinBox) {
        int frameWidth = pixelMetric(PM_DefaultFrameWidth, option,
                                     widget);
        int buttonWidth = 16;

        switch (whichSubControl) {
        case SC_SpinBoxFrame:
            return option->rect;
        case SC_SpinBoxEditField:
            return option->rect.adjusted(+buttonWidth, +frameWidth,
                                         -buttonWidth, -frameWidth);
        case SC_SpinBoxDown:
```

```
            return visualRect(option->direction, option->rect,
                              QRect(option->rect.x(), option->rect.y(),
                                    buttonWidth,
                                    option->rect.height()));
        case SC_SpinBoxUp:
            return visualRect(option->direction, option->rect,
                              QRect(option->rect.right() - buttonWidth,
                                    option->rect.y(),
                                    buttonWidth,
                                    option->rect.height()));
        default:
            return QRect();
        }
    } else {
        return QWindowsStyle::subControlRect(whichControl, option,
                                             whichSubControl, widget);
    }
}
```

Qt 调用 subControlRect()函数确定辅助控制器窗口部件的位置。例如，QSpinBox 调用它来确定在哪里放置 QLineEdit。处理鼠标事件时也用它查找被单击的辅助控制器窗口部件。此外，当实现 drawComplexControl()时，我们自己调用它，在 drawBronzeSpinBoxButton()中也会调用它。

在实现中，我们检查当前的窗口部件是否为微调框。如果确实如此，就返回这些窗口部件有意义的矩形区域、微调框的边框、编辑区域、向下按钮和向上按钮。图 19.20 显示了这些矩形区域之间的关系。对于其他的窗口部件，包括 QPushButton，依赖于它的基类的实现方法。

SC_SpinBoxDown 和 SC_SpinBoxUp 通过 QStyle::visualRect()返回矩形区域。调用 visualRect()使用以下的语法：

```
visualRect(direction, outerRect, logicalRect)
```

如果 *direction* 是 Qt::LeftToRight，则 *logicalRect* 将被原样返回；否则，*logicalRect* 针对 *outerRect* 进行翻转。这保证了图形元素从右到左排列，主要应用于像阿拉伯和希伯来这样的语言。对于对称的元素，例如 SC_SpinBoxFrame 和 SC_SpinBoxEditField，翻转没有意义，因此不必调用 visualRect()。测试从右到左样式，可以简单地在命令行加入-reverse 参数运行使用这一模式的应用程序。图 19.21 显示了从右到左模式下的 Bronze 样式。

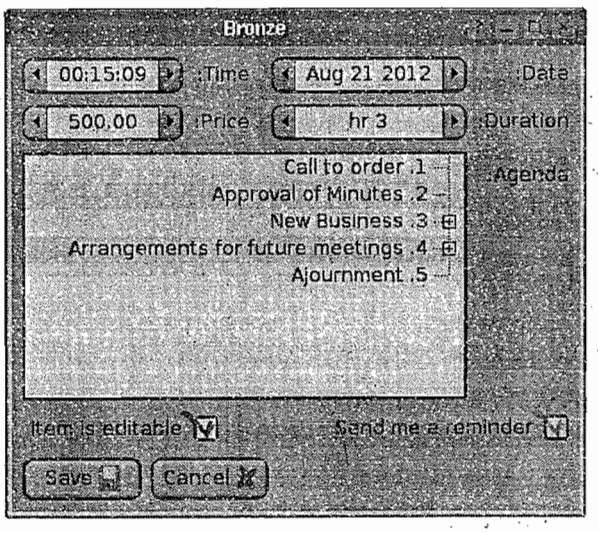


图 19.21　从右到左模式下的 Bronze 风格

这就完成了重新实现的 QWindowsStyle 的公有函数的介绍。下面的 4 个函数是私有绘图函数。

```
void BronzeStyle::drawBronzeFrame(const QStyleOption *option,
                                  QPainter *painter) const
{
    painter->save();
    painter->setRenderHint(QPainter::Antialiasing, true);
    painter->setPen(QPen(option->palette.foreground(), 1.0));
    painter->drawRect(option->rect.adjusted(+1, +1, -1, -1));
    painter->restore();
}
```

drawBronzeFrame()函数会被 drawPrimitive()调用,用来绘制一个 PE_Frame 私有元素。当边框形状为 QFrame::StyledPanel 时,它用来绘制 QFrame(或者其子类,例如 QTreeView)的边框。(其他的边框形状,例如 Box、Panel 和 VLine,直接由 QFrame 绘制,不使用样式。)

我们绘制的边框是 1 像素宽反走样处理的窗口部件的边框,使用调色板的前景画刷实现(可以使用 QStyleOption 的 palette 成员变量)。因为矩形是反走样的,位于整型坐标中,有 2 像素宽的模糊边框效果,正好是 Bronze 样式所需要的效果。

为了保证 QPainter 保存原来的状态,我们在调用 setRenderHint()和 setPen()之前调用 save(),最后调用 restore()恢复。这是必要的,因为 Qt 优化了绘图,重用同一个 QPainter 绘制几个图形元素。

下一个将要介绍的函数是 drawBronzeBevel(),可以用来绘制 QPushButton 的背景:

```
void BronzeStyle::drawBronzeBevel(const QStyleOption *option,
                                  QPainter *painter) const
{
    QColor buttonColor = option->palette.button().color();
    int coeff = (option->state & State_MouseOver) ? 115 : 105;

    QLinearGradient gradient(0, 0, 0, option->rect.height());
    gradient.setColorAt(0.0, option->palette.light().color());
    gradient.setColorAt(0.2, buttonColor.lighter(coeff));
    gradient.setColorAt(0.8, buttonColor.darker(coeff));
    gradient.setColorAt(1.0, option->palette.dark().color());
```

首先设置 QLinearGradient,它用来填充背景。渐变的上方是亮色,下方是暗色,其间是渐变的棕色。渐变点在 0.2 和 0.8,给按钮一个虚拟的三维效果。coeff 因子控制按钮的三维效果外观。当鼠标在按钮上移动时,coeff 因子使用 115%,使按钮有弹起效果。

```
    double penWidth = 1.0;
    if (const QStyleOptionButton *buttonOpt =
            qstyleoption_cast<const QStyleOptionButton *>(option)) {
        if (buttonOpt->features & QStyleOptionButton::DefaultButton)
            penWidth = 2.0;
    }
```

Bronze 样式对于默认按钮使用 2 像素宽的边框,否则使用 1 像素宽的边框。为了判断按钮是否为默认按钮,我们把 option 强制转换为 const QStyleOptionButton * 类型,检查其 features 成员变量。

```
    QRect roundRect = option->rect.adjusted(+1, +1, -1, -1);
    if (!roundRect.isValid())
        return;

    int diameter = 12;
    int cx = 100 * diameter / roundRect.width();
    int cy = 100 * diameter / roundRect.height();
```

我们定义了绘制按钮将要用到的更多的变量。cx 和 cy 系数指定按钮的圆角程度。它们根据 diameter 计算,指定圆角需要的直径。

```
    painter->save();
    painter->setPen(Qt::NoPen);
    painter->setBrush(gradient);
    painter->drawRoundRect(roundRect, cx, cy);
```

```
    if (option->state & (State_On | State_Sunken)) {
        QColor slightlyOpaqueBlack(0, 0, 0, 63);
        painter->setBrush(slightlyOpaqueBlack);
        painter->drawRoundRect(roundRect, cx, cy);
    }

    painter->setRenderHint(QPainter::Antialiasing, true);
    painter->setPen(QPen(option->palette.foreground(), penWidth));
    painter->setBrush(Qt::NoBrush);
    painter->drawRoundRect(roundRect, cx, cy);
    painter->restore();
}
```

最后执行绘图。首先使用函数提前定义好的 QLinearGradient 绘制背景。如果按钮当前是被按下的(或者是一个"选中"状态的切换按钮),则添加一个 75%透明的黑色效果,使它看起来稍暗一些。

一旦绘制了背景,我们打开反走样以获得平滑的圆角,设置一个合适的画笔,清空画刷,绘制边框。

```
void BronzeStyle::drawBronzeSpinBoxButton(SubControl which,
        const QStyleOptionComplex *option, QPainter *painter) const
{
    PrimitiveElement arrow = PE_IndicatorArrowLeft;
    QRect buttonRect = option->rect;
    if ((which == SC_SpinBoxUp)
            != (option->direction == Qt::RightToLeft)) {
        arrow = PE_IndicatorArrowRight;
        buttonRect.translate(buttonRect.width() / 2, 0);
    }
    buttonRect.setWidth((buttonRect.width() + 1) / 2);

    QStyleOption buttonOpt(*option);

    painter->save();
    painter->setClipRect(buttonRect, Qt::IntersectClip);
    if (!(option->activeSubControls & which))
        buttonOpt.state &= ~(State_MouseOver | State_On | State_Sunken);
    drawBronzeBevel(&buttonOpt, painter);

    QStyleOption arrowOpt(buttonOpt);
    arrowOpt.rect = subControlRect(CC_SpinBox, option, which)
                    .adjusted(+3, +3, -3, -3);
    if (arrowOpt.rect.isValid())
        drawPrimitive(arrow, &arrowOpt, painter);
    painter->restore();
}
```

drawBronzeSpinBoxButton()函数绘制微调框的向上或者向下的按钮,这依赖于 which 的值是 SC_SpinBoxDown 还是 SC_SpinBoxUp。我们首先设置按钮上面的箭头(向左的或向右的箭头),以及绘制按钮的矩形。

如果 which 的值是 SC_SpinBoxDown(或者 which 的值是 SC_SpinBoxUp 并且布局的方向是从右到左),则使用向左的箭头(PE_IndicatorArrowLeft),并且在微调框矩形的左半边绘制按钮;否则,使用向右的箭头,在右半边绘制按钮。

为了绘制按钮,使用能够正确反映微调框状态的 QStyleOption 调用 drawBronzeBevel()。例如,如果鼠标在微调框上方移动,但并非 which 指定的微调框按钮,我们就从 QStyleOption 的状态中清除 State_MouseOver、State_On 和 State_Suoken 标识,这样可以保证两个微调框按钮可以互不干扰。

在执行绘制之前,我们调用 setClipRect()设置 QPainter 上的剪切矩形。这是因为我们只想绘制按钮斜面的左半边或者右半边,不是整个按钮斜面。

最后,在结尾处调用 drawPrimitive()绘制箭头。用来绘制箭头的 QStyleOption 提供一个矩形,这个矩形与微调框按钮矩形(SC_SpinBoxUp 或者 SC_SpinBoxDown)相对应,但稍小一些,以获取小型箭头。

```
void BronzeStyle::drawBronzeCheckBoxIndicator(
        const QStyleOption *option, QPainter *painter) const
{
    painter->save();
    painter->setRenderHint(QPainter::Antialiasing, true);

    if (option->state & State_MouseOver) {
        painter->setBrush(option->palette.alternateBase());
    } else {
        painter->setBrush(option->palette.base());
    }
    painter->drawRoundRect(option->rect.adjusted(+1, +1, -1, -1));

    if (option->state & (State_On | State_NoChange)) {
        QPixmap pixmap;
        if (!(option->state & State_Enabled)) {
            pixmap.load(":/images/checkmark-disabled.png");
        } else if (option->state & State_NoChange) {
            pixmap.load(":/images/checkmark-partial.png");
        } else {
            pixmap.load(":/images/checkmark.png");
        }

        QRect pixmapRect = pixmap.rect()
                           .translated(option->rect.topLeft())
                           .translated(+2, -6);
        QRect painterRect = visualRect(option->direction, option->rect,
                                       pixmapRect);
        if (option->direction == Qt::RightToLeft) {
            painter->scale(-1.0, +1.0);
            painterRect.moveLeft(-painterRect.right() - 1);
        }
        painter->drawPixmap(painterRect, pixmap);
    }
    painter->restore();
}
```

尽管 drawBronzeCheckBoxIndicator()的代码起初看起来有些复杂,绘制复选框的指示器实际上相当简单:使用 drawRoundRect()绘制一个矩形,并且使用 drawPixmap()绘制选取标志。复杂度提高是因为当鼠标在复选框指示器上移动时我们想使用一种不同的背景色,我们区分启用标志、禁用标志以及未决标志(三状态复选框),并在从右到左模式下翻转选取标志(通过翻转 QPainter 的坐标系统)。

QStyleOption 版本

QStyle 在绘制窗口部件时所需的信息通过 QStyleOption 和它的子类(QStyleOptionButton、QStyleOptionComBox、QStyleOptionFrame 等)来提供。为了提高性能,数据保存在公有成员变量中。

为了保证所有 Qt 4 版本的二进制兼容性,Trolltech 不能为这些类添加新的成员变量,只有到 Qt 5 时才可以这样做。为了增强 Qt 4.*x* 发行版的功能,QStyleOption 拥有一个 version 变量,用以区分同一个类的不同版本。当需要新的数据成员时,Trolltech 才把它们添加到一个相同类型但具有不同 version 的子类中。例如,Qt 4.1 发布了 QStyleOptionFrameV2,QStyleOptionFrameV2 从 QStyleOptionFrame 继承,提供一个可供查询的 features 成员变量。QStyleOptionFrameV2 子类是 SO_Frame 类型的,但其 version 是 2,而不是 1。

在 QStyle 子类中,通常可以使用 QStyleOptionFrame,但如果想访问只在版本 2 中定义的 features 变量,可以这样实现:

```
if (const QStyleOptionFrame *frameOption =
        qstyleoption_cast<const QStyleOptionFrame *>(option)) {
    QStyleOptionFrameV2 frameOptionV2(*frameOption);

    int lineWidth = frameOptionV2.lineWidth;
    bool flat = (frameOptionV2.features & QStyleOptionFrameV2::Flat);
    ...
}
```

QStyleOptionFrameV2 的副本构造函数可以接受两个版本的类的实例。如果提供的是版本 1 的对象，构造函数将用默认值 None 初始化 features 字段；否则，它将从 frameOption 对象中复制 features 字段。

不用副本而要达到此目的的另外一种办法是使用 qstyleoption_cast <T> () 来区分这些不同的版本：

```
if (const QStyleOptionFrame *frameOption =
        qstyleoption_cast<const QStyleOptionFrame *>(option)) {
    int lineWidth = frameOption.lineWidth;
    bool flat = false;

    if (const QStyleOptionFrame *frameOptionV2 =
            qstyleoption_cast<const QStyleOptionFrameV2 *>(option))
        flat = (frameOptionV2.features & QStyleOptionFrameV2::Flat);
    ...
}
```

在 Qt 5 中，features 变量很可能被移到 QStyleOptionFrame 类中，因为 Qt 不提供主要发布版本之间的二进制兼容性。

现在，我们就完成了 Bronze QStyle 子类的实现。在图 19.17 中，QDateEdit 和 QTreeWidget 都使用了 Bronze 样式，尽管没有编写针对它们的代码。这是因为 QDateEdit、QDoubleSpinBox 以及其他一些窗口部件都是“微调框”，因此可以使用 Bronze 样式代码绘制它们。同样，QTreeWidget 以及其他的一些继承自 QFrame 的窗口部件也可以使用 Bronze 样式自定义外观。

本节中介绍的 Bronze 样式可以很容易地应用于应用程序中，只需把它连接进去，然后在应用程序的 main()函数中调用：

```
QApplication::setStyle(new BronzeStyle);
```

就可以了。没有被 Bronze 样式处理的窗口部件则会保持传统的 Windows 外观。自定义样式也可以被编译成插件，在高级版本的 Qt 设计师中使用，使用该样式预览窗体。第 21 章将介绍如何将 Bronze 样式编译成 Qt 的插件。

尽管这里开发的样式仅有大约 300 行的代码，但值得注意的是，要开发一种功能齐全的自定义样式则是一项很大的工程，一般需要 3000 到 5000 行的 C++ 代码。因为这个原因，在可能的情况下，使用 Qt 样式表则显得更为容易和方便，或者使用两者混合的方法，将样式表和自定义样式结合起来使用。如果你计划创建一个自定义样式，实现样式以及样式化窗口部件的方法在网站 http://doc.trolltech.com/4.3/style-reference.html 上有深入的介绍。

第 20 章　三维图形

OpenGL 是绘制三维图形的标准 API。Qt 应用程序可以使用 QtOpenGL 模块绘制三维图形，该模块依赖于系统的 OpenGL 库。Qt OpenGL 模块提供 QGLWidget 类，可以通过对它的子类化，并使用 OpenGL 命令开发出自己的窗口部件。对于许多三维应用程序来说，这就足够了。本章第一节介绍了一个简单的应用程序，该程序使用这一技术绘制一个三角锥，用户可以使用鼠标与其交互。

从 Qt 4 开始，可以在 QGLWidget 上使用 QPainter，尽管它还只是一个普通的窗口部件。在 QGLWidget 上使用 QPainter 的好处是可以使用 OpenGL 高效地绘图，例如进行坐标变换以及像素映射绘制。使用 QPainter 的另一个好处是可以使用其二维的高级 API，并结合 OpenGL 调用来绘制三维图形。本章第二节将介绍如何在同一窗口部件中使用 QPainter 和 OpenGL 命令绘制二维和三维的组合。

使用 QGLWidget，可以在场景中使用 OpenGL 作为后端绘制三维场景。为了在硬件加速的离屏表面上绘制，可以使用 pbuffer 和 framebuffer 对象进行扩展，它们分别在 QGLPixelBuffer 和 QGLFramebufferObject 类中。本章的第三节将介绍如何使用帧缓存对象制作叠加。

本章假设读者已熟悉 OpenGL。如果刚开始接触 OpenGL，那么 http://www.opengl.org/是学习它的好地方。

20.1　使用 OpenGL 绘图

在 Qt 应用程序中绘制 OpenGL 是很简单的：必须子类化 QGLWidget，实现几个虚函数，连接 QtOpenGL 和 OpenGL 库。由于 QGLWidget 派生自 QWidget，许多已知的技术仍然适用。最大的区别是要使用标准的 OpenGL 函数替代 QPainter 来实现绘制。

为了介绍 QGLWidget 的用法，我们将查看图 20.1 所示的 Tetrahedron 应用程序的代码。该应用程序可以显示一个三维的三角锥(tetrahedron)，或者也可以称为四面体，它的每个面都具有不同的颜色。用户可以通过单击并拖动鼠标来旋转它。用户也可以通过双击某个面，并从弹出的 QColorDialog 中选择一种颜色来设置这个面的颜色。

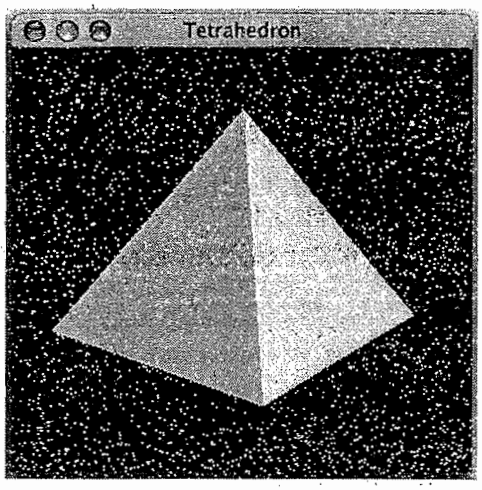


图 20.1　Tetrahedron 应用程序

```
class Tetrahedron : public QGLWidget
{
    Q_OBJECT

public:
    Tetrahedron(QWidget *parent = 0);

protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseDoubleClickEvent(QMouseEvent *event);

private:
    void draw();
    int faceAtPosition(const QPoint &pos);

    GLfloat rotationX;
    GLfloat rotationY;
    GLfloat rotationZ;
    QColor faceColors[4];
    QPoint lastPos;
};
```

Tetrahedron 类从 QGLWidget 中派生而来。initializeGL()、resizeGL()和 paintGL()这三个函数是在 QGLWidget 中实现的。鼠标事件处理器在 QWidget 中实现。

```
Tetrahedron::Tetrahedron(QWidget *parent)
    : QGLWidget(parent)
{
    setFormat(QGLFormat(QGL::DoubleBuffer | QGL::DepthBuffer));

    rotationX = -21.0;
    rotationY = -57.0;
    rotationZ = 0.0;
    faceColors[0] = Qt::red;
    faceColors[1] = Qt::green;
    faceColors[2] = Qt::blue;
    faceColors[3] = Qt::yellow;
}
```

在构造函数中,调用 QGLWidget::setFormat()指定 OpenGL 的显示描述表,并且初始化私有变量。

```
void Tetrahedron::initializeGL()
{
    qglClearColor(Qt::black);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}
```

initializeGL()函数在调用 paintGL()之前只被调用一次。可以在这里设置 OpenGL 的绘图描述表,定义显示列表,以及执行其他的初始化。

所有的代码都是标准的 OpenGL,除了对 QGLWidget 的 qglClearColor()函数的调用。如果想坚持使用标准的 OpenGL,则可以在 RGBA 模式下调用 glClearColor(),而在颜色索引模式下调用 glClearIndex()。

```
void Tetrahedron::resizeGL(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    GLfloat x = GLfloat(width) / height;
    glFrustum(-x, +x, -1.0, +1.0, 4.0, 15.0);
    glMatrixMode(GL_MODELVIEW);
}
```

应在第一次调用 paintGL()之前,但在 initializeGL()之后调用 resizeGL()函数。在窗口部件改变大小时也将调用 resizeGL()函数。在这里可以设置 OpenGL 视口、投影以及其他与窗口部件尺寸相关的设置。

```
void Tetrahedron::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    draw();
}
```

在窗口部件需要重绘时调用 paintGL()函数。它与 QWidget::paintEvent()类似,使用 OpenGL 函数代替 QPainter 函数。实际的绘制由私有函数 draw()实现。

```
void Tetrahedron::draw()
{
    static const GLfloat P1[3] = { 0.0, -1.0, +2.0 };
    static const GLfloat P2[3] = { +1.73205081, -1.0, -1.0 };
    static const GLfloat P3[3] = { -1.73205081, -1.0, -1.0 };
    static const GLfloat P4[3] = { 0.0, +2.0, 0.0 };

    static const GLfloat * const coords[4][3] = {
        { P1, P2, P3 }, { P1, P3, P4 }, { P1, P4, P2 }, { P2, P4, P3 }
    };

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(rotationX, 1.0, 0.0, 0.0);
    glRotatef(rotationY, 0.0, 1.0, 0.0);
    glRotatef(rotationZ, 0.0, 0.0, 1.0);

    for (int i = 0; i < 4; ++i) {
        glLoadName(i);
        glBegin(GL_TRIANGLES);
        qglColor(faceColors[i]);
        for (int j = 0; j < 3; ++j) {
            glVertex3f(coords[i][j][0], coords[i][j][1],
                       coords[i][j][2]);
        }
        glEnd();
    }
}
```

在 draw()中,我们绘制三角锥,应用 x、y 和 z 轴的旋转以及保存在 faceColors 数组中的颜色。每个调用都是标准的 OpenGL,除了 qglColor()函数。可以根据颜色模式使用 OpenGL 函数 glColor 3d()或者 glIndex()。

```
void Tetrahedron::mousePressEvent(QMouseEvent *event)
{
    lastPos = event->pos();
}

void Tetrahedron::mouseMoveEvent(QMouseEvent *event)
{
    GLfloat dx = GLfloat(event->x() - lastPos.x()) / width();
    GLfloat dy = GLfloat(event->y() - lastPos.y()) / height();
    if (event->buttons() & Qt::LeftButton) {
        rotationX += 180 * dy;
        rotationY += 180 * dx;
        updateGL();
    } else if (event->buttons() & Qt::RightButton) {
        rotationX += 180 * dy;
        rotationZ += 180 * dx;
        updateGL();
    }
    lastPos = event->pos();
}
```

mousePressEvent()和 mouseMoveEvent()函数在 QWidget 中实现,用户可以通过单击并拖动鼠标旋转视图。鼠标左键可以沿着 x 和 y 轴旋转,右键可以沿着 x 和 z 轴旋转。

当 rotationX、rotationY 和 rotationZ 其中一个变量的值改变时,我们调用 updateGL()重绘场景。

```
void Tetrahedron::mouseDoubleClickEvent(QMouseEvent *event)
{
    int face = faceAtPosition(event->pos());
    if (face != -1) {
        QColor color = QColorDialog::getColor(faceColors[face], this);
        if (color.isValid()) {
            faceColors[face] = color;
            updateGL();
        }
    }
}
```

mouseDoubleClickEvent()在 QWidget 中实现,用户可以双击三角锥的面设置其颜色。调用私有函数 faceAtPosition()确定光标下的面。如果用户双击了某个面,就调用 QColorDialog::getColor(),为该面获取一种新的颜色。然后,使用新颜色更新 faceColors 数组,并且调用 updateGL()重绘场景。

```
int Tetrahedron::faceAtPosition(const QPoint &pos)
{
    const int MaxSize = 512;
    GLuint buffer[MaxSize];
    GLint viewport[4];

    makeCurrent();

    glGetIntegerv(GL_VIEWPORT, viewport);
    glSelectBuffer(MaxSize, buffer);
    glRenderMode(GL_SELECT);

    glInitNames();
    glPushName(0);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluPickMatrix(GLdouble(pos.x()), GLdouble(viewport[3] - pos.y()),
                  5.0, 5.0, viewport);
    GLfloat x = GLfloat(width()) / height();
    glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
    draw();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

    if (!glRenderMode(GL_RENDER))
        return -1;
    return buffer[3];
}
```

faceAtPosition()函数返回窗口部件某位置所处的面的编号,如果没有面就返回 -1。确定被选面的代码在 OpenGL 中有点复杂。事实上,我们以 GL_SELECT 模式绘制场景,使用 OpenGL 的选择功能,从 OpenGL 的命中记录中读取面编号(它的"名字")。代码是标准的 OpenGL 代码,除了起初的 QGLWidget::makeCurrent()调用,它可以确保我们正确地使用 OpenGL 描述表。[QGLWidget 在调用 initializeGL()、resizeGL()或 paintGL()之前会自动调用它,在 Tetrahedron 实现的其他地方则不需要调用它。]

以下是该应用程序的 main()函数:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    if (!QGLFormat::hasOpenGL()) {
```

```
        std::cerr << "This system has no OpenGL support" << std::endl;
        return 1;
    }

    Tetrahedron tetrahedron;
    tetrahedron.setWindowTitle(QObject::tr("Tetrahedron"));
    tetrahedron.resize(300, 300);
    tetrahedron.show();

    return app.exec();
}
```

如果用户的系统不支持 OpenGL,则会在控制台打印错误信息并立即返回。

为了使应用程序正确连接 QtOpenGL 模块和系统的 OpenGL 库,需要在 .pro 文件中添加这一项:

```
QT += opengl
```

这就完成了 Tetrahedron 应用程序。

20.2 OpenGL 和 QPainter 的结合

前一节中已经看到了如何在 QGLWidget 上使用 OpenGL 命令绘制一个三维场景。我们也可以使用 QPainter 在 QGLWidget 上绘制二维图形。本节的 Vowel Cube 例子组合使用了 OpenGL 和 QPainter,以便尽可能地做到两全其美。本节还给出了 QGLWidget::renderText()函数的用法,该函数可以在三维场景中绘制不变形的文本标注。该应用程序的运行效果如图 20.2 所示。

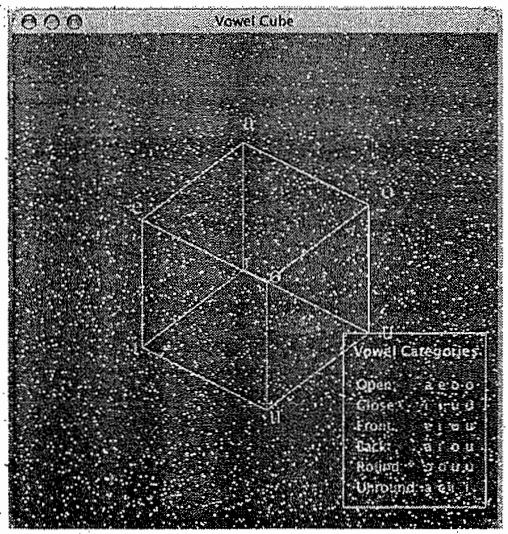


图 20.2 Vowel Cube 应用程序

Vowel Cube 显示一个由土耳其语言的 8 个元音字母组成的立方体,这一图形经常出现在介绍土耳其语法和语言的书中。在前景部分,一个图例列举了元音的类别,以及各个元音所属的类别。这个立方体使这一信息更为直观。比如说,前元音位于立方体的前面,后元音位于立方体的后面。背景部分则使用了辐射渐变。

```
class VowelCube : public QGLWidget
{
    Q_OBJECT

public:
```

```
    VowelCube(QWidget *parent = 0);
    ~VowelCube();

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void wheelEvent(QWheelEvent *event);

private:
    void createGradient();
    void createGLObject();
    void drawBackground(QPainter *painter);
    void drawCube();
    void drawLegend(QPainter *painter);

    GLuint glObject;
    QRadialGradient gradient;
    GLfloat rotationX;
    GLfloat rotationY;
    GLfloat rotationZ;
    GLfloat scaling;
    QPoint lastPos;
};
```

VowelCube 类派生自 QGLWidget,它使用 QPainter 绘制背景的渐变,然后使用 OpenGL 调用绘制立方体,接着使用 renderText()绘制立方体角上的 8 个元音字母,最后使用 QPainter 和 QTextDocument 绘制图例。用户可以单击并拖动鼠标来旋转立方体,并且可以使用鼠标滚轮进行放大或缩小。

不像前面一节中三角锥的例子,在那个例子中,我们实现了 QGLWidget 的高级函数 initializeGL()、resizeGL()和 paintGL(),这次我们实现传统的 QWidget 处理,这更有利于更新 OpenGL 的帧缓存。

```
VowelCube::VowelCube(QWidget *parent)
    : QGLWidget(parent)
{
    setFormat(QGLFormat(QGL::SampleBuffers));

    rotationX = -38.0;
    rotationY = -58.0;
    rotationZ = 0.0;
    scaling = 1.0;

    createGradient();
    createGLObject();
}
```

在构造函数中,首先调用 QGLWidget::setFormat()使 OpenGL 的显示描述表支持反走样。然后初始化类的各个私有变量。最后,我们调用 createGradient()设置用来填充背景的 QRadialGradient,并且调用 createGLObject()创建 OpenGL 立方体对象。通过构造函数中的这些工作,我们稍后可以更迅捷地重绘场景。

```
void VowelCube::createGradient()
{
    gradient.setCoordinateMode(QGradient::ObjectBoundingMode);
    gradient.setCenter(0.45, 0.50);
    gradient.setFocalPoint(0.40, 0.45);
    gradient.setColorAt(0.0, QColor(105, 146, 182));
    gradient.setColorAt(0.4, QColor(81, 113, 150));
    gradient.setColorAt(0.8, QColor(16, 56, 121));
}
```

在 createGradient()中,我们简单地使用不同的蓝色渐变色设置 QRadialGradient。setCoordinateMode()调用确保指定的中心和焦点坐标根据窗口部件大小进行调整。其位置用 0 和 1 之间的浮点数表示,0 对应焦点坐标,1 对应圆的边缘。

```
void VowelCube::createGLObject()
{
    makeCurrent();

    glShadeModel(GL_FLAT);

    glObject = glGenLists(1);
    glNewList(glObject, GL_COMPILE);
    qglColor(QColor(255, 239, 191));
    glLineWidth(1.0);

    glBegin(GL_LINES);
    glVertex3f(+1.0, +1.0, -1.0);
    ...
    glVertex3f(-1.0, +1.0, +1.0);
    glEnd();

    glEndList();
}
```

createGLObject()创建 OpenGL 列表,该列表保存绘制的立方体的边。除了在开头部分的 QGLWidget::makeCurrent()调用外,这些代码都是标准的 OpenGL 代码,但它可以确保我们正确地使用 OpenGL 描述表。

```
VowelCube::~VowelCube()
{
    makeCurrent();
    glDeleteLists(glObject, 1);
}
```

在析构函数中,调用 glDeleteLists()删除构造函数创建的 OpenGL 立方体对象,然后需要再一次调用 makeCurrent()。

```
void VowelCube::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    drawBackground(&painter);
    drawCube();
    drawLegend(&painter);
}
```

在 paintEvent()中,像普通的 QWidget 一样来设置 QPainter,然后绘制背景、立方体以及图例。

```
void VowelCube::drawBackground(QPainter *painter)
{
    painter->setPen(Qt::NoPen);
    painter->setBrush(gradient);
    painter->drawRect(rect());
}
```

绘制背景,简单来说就是使用合适的画刷调用 drawRect()。

drawCube()函数是该定制窗口部件的核心。我们分两部分来介绍它:

```
void VowelCube::drawCube()
{
    glPushAttrib(GL_ALL_ATTRIB_BITS);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    GLfloat x = 3.0 * GLfloat(width()) / height();
    glOrtho(-x, +x, -3.0, +3.0, 4.0, 15.0);

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -10.0);
```

```
    glScalef(scaling, scaling, scaling);

    glRotatef(rotationX, 1.0, 0.0, 0.0);
    glRotatef(rotationY, 0.0, 1.0, 0.0);
    glRotatef(rotationZ, 0.0, 0.0, 1.0);

    glEnable(GL_MULTISAMPLE);
```

因为在两段代码中有一些 OpenGL 的代码，所以必须小心——特别地，我们必须保存修改的 OpenGL 状态，然后再在适当的时候恢复它。因此，需要保存 OpenGL 的属性、投影矩阵以及修改前的模式视图矩阵。最后，设置 GL_MULTISAMPLE 参数启动反走样。

```
    glCallList(glObject);

    setFont(QFont("Times", 24));
    qglColor(QColor(255, 223, 127));

    renderText(+1.1, +1.1, +1.1, QChar('a'));
    renderText(-1.1, +1.1, +1.1, QChar('e'));
    renderText(+1.1, +1.1, -1.1, QChar('o'));
    renderText(-1.1, +1.1, -1.1, QChar(0x00F6));
    renderText(+1.1, -1.1, +1.1, QChar(0x0131));
    renderText(-1.1, -1.1, +1.1, QChar('i'));
    renderText(+1.1, -1.1, -1.1, QChar('u'));
    renderText(-1.1, -1.1, -1.1, QChar(0x00FC));

    glMatrixMode(GL_MODELVIEW);
    glPopMatrix();

    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

    glPopAttrib();
}
```

接着，调用 glCallList()绘制立方体对象。然后设置字体和颜色，调用 QGLWidget::renderText()绘制立方体角上的元音。对于不能用 ASCII 表示的土耳其元音，就使用它们的 Unicode 值。

renderText()函数带一组(x,y,z)坐标，指定文字在模式视图上的位置。文字本身不能变形。

```
void VowelCube::drawLegend(QPainter *painter)
{
    const int Margin = 11;
    const int Padding = 6;

    QTextDocument textDocument;
    textDocument.setDefaultStyleSheet("* { color: #FFEFEF }");
    textDocument.setHtml("<h4 align=\"center\">Vowel Categories</h4>"
                         "<p align=\"center\"><table width=\"100%\">"
                         "<tr><td>Open:<td>a<td>e<td>o<td>&ouml;"
                         ...
                         "</table>");
    textDocument.setTextWidth(textDocument.size().width());

    QRect rect(QPoint(0, 0), textDocument.size().toSize()
                             + QSize(2 * Padding, 2 * Padding));
    painter->translate(width() - rect.width() - Margin,
                       height() - rect.height() - Margin);
    painter->setPen(QColor(255, 239, 239));
    painter->setBrush(QColor(255, 0, 0, 31));
    painter->drawRect(rect);

    painter->translate(Padding, Padding);
    textDocument.drawContents(painter);
}
```

在 drawLegend()函数中，我们使用一些 HTML 文字设置 QTextDocument 对象，列举了土耳其语言的元音种类和元音，在半透明的蓝色矩形上绘制它们。

VowelCube 窗口部件实现了 mousePressEvent()、mouseMoveEvent()以及 wheelEvent(),但没有什么特别之处。如同在一个标准的 Qt 定制组件中,在需要重绘的时候调用 update()。例如,以下是 wheelEvent()的代码:

```
void VowelCube::wheelEvent(QWheelEvent *event)
{
    double numDegrees = -event->delta() / 8.0;
    double numSteps = numDegrees / 15.0;
    scaling *= std::pow(1.125, numSteps);
    update();
}
```

这样就完成了对本例的介绍。在 VowelCube 的 paintEvent()实现中,使用以下通用模式:

1. 创建一个 QPainter。
2. 使用 QPainter 绘制背景。
3. 保存 OpenGL 状态。
4. 使用 OpenGL 操作绘制场景。
5. 恢复 OpenGL 状态。
6. 使用 QPainter 绘制前景。
7. 销毁 QPainter。

还有其他的可能。例如,如果不绘制背景,可以这样做:

1. 使用 OpenGL 操作绘制场景。
2. 创建 QPainter。
3. 使用 QPainter 绘制前景。
4. 销毁 QPainter。

这对应于下面的代码:

```
void VowelCube::paintEvent(QPaintEvent * /* event */)
{
    drawCube();
    drawLegend();
}

void VowelCube::drawCube()
{
    ...
}

void VowelCube::drawLegend()
{
    QPainter painter(this);
    ...
}
```

值得注意的是,这一次我们在 drawLegend()函数中创建 QPainter 对象。该方法主要的好处是不需要保存和恢复 OpenGL 的状态。但不能就这样使用它,因为 QPainter 在开始绘制之前会自动清除背景,从而覆盖 OpenGL 场景。为了避免这种情况,必须在窗口部件的构造函数中调用 setAutoFillBackground(false)。

如果只绘制背景和立方体,而不绘制前景,那么就会发生另外一种有趣的模式:

1. 创建 QPainter。
2. 使用 QPainter 绘制背景。

3. 销毁 QPainter。

4. 使用 OpenGL 操作绘制场景。

这次不需要保存和恢复 OpenGL 状态。但这样也不能工作,因为 QPainter 为了使结果可见在其析构函数中自动调用 QGLWidget::swapBuffers(),任何 QPainter 析构函数之后的 OpenGL 调用将会作用于离屏缓存,而不会在屏幕上显示。为了避免这一点,必须在窗口部件的构造函数中调用 setAutoBufferSwap(false),在 paintEvent()的结尾处调用 swapBuffer()。例如:

```
void VowelCube::paintEvent(QPaintEvent * /* event */)
{
    drawBackground();
    drawCube();
    swapBuffers();
}

void VowelCube::drawBackground()
{
    QPainter painter(this);
    ...
}

void VowelCube::drawCube()
{
    ...
}
```

总之,最通用的方法是在 paintEvent()中创建一个 QPainter,在进行纯 OpenGL 操作时保存和恢复其状态。倘若记住以下几点,就可以避免一些状态的保存操作:

- QPainter 的构造函数[或者 QPainter::begin()]自动调用 glClear(),除非预先调用窗口部件的 setAutoFillBackground(false)。
- QPainter 的析构函数[或者 QPainter::end()]自动调用 QGLWidget::swapBuffers()切换可见缓存和离屏缓存,除非预先调用窗口部件的 setAutoBufferSwap(false)。
- 当 QPainter 被激活,我们可以交叉使用纯 OpenGL 命令,只要在执行纯 OpenGL 命令之前保存 OpenGL 状态,之后恢复 OpenGL 状态。

注意到这几点,将 OpenGL 和 QPainter 结合起来就变得容易多了,QPainter 和 OpenGL 为我们提供了更强的图形处理能力。

20.3 使用帧缓存对象生成叠加

通常,我们需要在一个复杂的三维场景中绘制简单的注解。如果场景非常复杂,可能需要几秒钟来绘制它。为了避免重复绘制场景,每当注解改变时,我们可以使用 X11 叠加或者 OpenGL 内置支持的叠加。

最近,pbuffers 和 framebuffer 对象提供了更方便和更灵活的制作叠加的方法。基本的思路是在离屏表面绘制三维场景,并将其绑定纹理。纹理通过绘制矩形映射到屏幕,注解绘制在上面。当注解改变时,只需重绘矩形和注解。概念上,这类似于第 5 章的二维 Plotter 窗口部件。

为了阐述这一技术,我们将介绍如图 20.3 所示的 Teapots 应用程序。该应用程序由单个 OpenGL 窗口组成,该窗口显示了一组茶壶,用户可以在上面单击并拖动鼠标来绘制橡皮筋选择框。茶壶不以任何方式移动或改变,除非窗口大小发生变化。实现上依靠 framebuffer 对象来保存茶壶的场景。也可以使用 pbuffer 来实现相同的效果,用 QGLPixelBuffer 代替 QGLFramebufferObject。

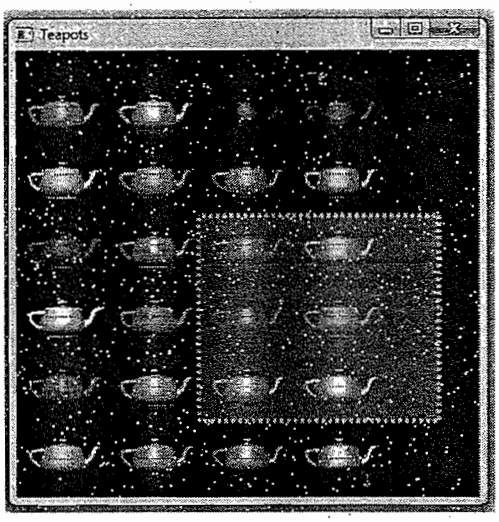


图 20.3 Teapots 应用程序

```
class Teapots : public QGLWidget
{
    Q_OBJECT

public:
    Teapots(QWidget *parent = 0);
    ~Teapots();

protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);

private:
    void createGLTeapotObject();
    void drawTeapot(GLfloat x, GLfloat y, GLfloat ambientR,
                    GLfloat ambientG, GLfloat ambientB,
                    GLfloat diffuseR, GLfloat diffuseG,
                    GLfloat diffuseB, GLfloat specularR,
                    GLfloat specularG, GLfloat specularB,
                    GLfloat shininess);
    void drawTeapots();
    QGLFramebufferObject *fbObject;
    GLuint glTeapotObject;
    QPoint rubberBandCorner1;
    QPoint rubberBandCorner2;
    bool rubberBandIsShown;
};
```

Teapots 类派生自 QGLWidget,实现了 OpenGL 的高级处理函数 initializeGL()、resizeGL()和 paintGL()。同时,它还实现了 mousePressEvent()、mouseMoveEvent()和 mouseReleaseEvent(),以便可以让用户绘制一个橡皮筋选择框。

私有函数负责创建茶壶对象以及茶壶的绘制。代码基于"OpenGL Programming Guide"这本书中的 teapots 例子,而且很复杂,该书的作者是 Jackie Neider、Tom Davis 和 Mason Woo(Addison-Wesley, 1993)。由于它跟我们的目的没有直接的关系,所以在这里就不再介绍了。

这些私有变量可以保存 framebuffer 对象、茶壶对象、橡皮筋选择框的各个角,以及橡皮筋选择框的可见属性。

```
Teapots::Teapots(QWidget *parent)
    : QGLWidget(parent)
{
    rubberBandIsShown = false;

    makeCurrent();
    fbObject = new QGLFramebufferObject(1024, 1024,
                                        QGLFramebufferObject::Depth);
    createGLTeapotObject();
}
```

Teapots 构造函数初始化 rubberBandIsShown 私有变量，创建 framebuffer 对象，并创建茶壶对象。我们将跳过 createGLTeapotObject()函数，因为它很长而且并不包含与 Qt 相关的代码。

```
Teapots::~Teapots()
{
    makeCurrent();
    delete fbObject;
    glDeleteLists(glTeapotObject, 1);
}
```

在析构函数中，释放与 framebuffer 对象和茶壶相关的资源。

```
void Teapots::initializeGL()
{
    static const GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
    static const GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
    static const GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };
    static const GLfloat lmodelAmbient[] = { 0.2, 0.2, 0.2, 1.0 };
    static const GLfloat localView[] = { 0.0 };

    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodelAmbient);
    glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, localView);

    glFrontFace(GL_CW);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_NORMALIZE);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);
}
```

initializeGL()函数用来设置光照模式，打开各种 OpenGL 特性。代码直接来自于前面提到的“OpenGL Programming Guide”这本书的 teapots 例子。

```
void Teapots::resizeGL(int width, int height)
{
    fbObject->bind();

    glDisable(GL_TEXTURE_2D);
    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);

    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (width <= height) {
        glOrtho(0.0, 20.0, 0.0, 20.0 * GLfloat(height) / GLfloat(width),
                -10.0, 10.0);
    } else {
        glOrtho(0.0, 20.0 * GLfloat(width) / GLfloat(height), 0.0, 20.0,
                -10.0, 10.0);
    }
    glMatrixMode(GL_MODELVIEW);
    drawTeapots();

    fbObject->release();
}
```

resizeGL()函数在 Teapot 窗口部件改变大小时重绘茶壶场景。为了在 framebuffer 对象上绘制茶壶,在函数的开头调用 QGLFramebufferObject::bind()。然后,设置一些 OpenGL 特性,以及投影和模式视图矩阵。结尾的 drawTeapots()调用在 framebuffer 对象上绘制茶壶。最后,release()调用释放 framebuffer 对象,确保随后的 OpenGL 绘制操作不会发生在 framebuffer 对象上。

```
void Teapots::paintGL()
{
    glDisable(GL_LIGHTING);
    glViewport(0, 0, width(), height());
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glDisable(GL_DEPTH_TEST);

    glClear(GL_COLOR_BUFFER_BIT);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, fbObject->texture());
    glColor3f(1.0, 1.0, 1.0);
    GLfloat s = width() / GLfloat(fbObject->size().width());
    GLfloat t = height() / GLfloat(fbObject->size().height());

    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(-1.0, -1.0);
    glTexCoord2f(s, 0.0);
    glVertex2f(1.0, -1.0);
    glTexCoord2f(s, t);
    glVertex2f(1.0, 1.0);
    glTexCoord2f(0.0, t);
    glVertex2f(-1.0, 1.0);
    glEnd();
```

在 paintGL()函数中,首先重新设置投影和模式视图矩阵。然后,把 framebuffer 对象绑定到纹理上,绘制一个带有该纹理的矩形覆盖整个窗口部件。

```
    if (rubberBandIsShown) {
        glMatrixMode(GL_PROJECTION);
        glOrtho(0, width(), height(), 0, 0, 100);
        glMatrixMode(GL_MODELVIEW);
        glDisable(GL_TEXTURE_2D);
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glLineWidth(4.0);
        glColor4f(1.0, 1.0, 1.0, 0.2);
        glRecti(rubberBandCorner1.x(), rubberBandCorner1.y(),
                rubberBandCorner2.x(), rubberBandCorner2.y());
        glColor4f(1.0, 1.0, 0.0, 0.5);
        glLineStipple(3, 0xAAAA);
        glEnable(GL_LINE_STIPPLE);

        glBegin(GL_LINE_LOOP);
        glVertex2i(rubberBandCorner1.x(), rubberBandCorner1.y());
        glVertex2i(rubberBandCorner2.x(), rubberBandCorner1.y());
        glVertex2i(rubberBandCorner2.x(), rubberBandCorner2.y());
        glVertex2i(rubberBandCorner1.x(), rubberBandCorner2.y());
        glEnd();

        glLineWidth(1.0);
        glDisable(GL_LINE_STIPPLE);
        glDisable(GL_BLEND);
    }
}
```

如果橡皮筋选择框是可见的,我们可以把它绘制在矩形上。代码是标准的 OpenGL。

```
void Teapots::mousePressEvent(QMouseEvent *event)
{
    rubberBandCorner1 = event->pos();
    rubberBandCorner2 = event->pos();
    rubberBandIsShown = true;
}

void Teapots::mouseMoveEvent(QMouseEvent *event)
{
    if (rubberBandIsShown) {
        rubberBandCorner2 = event->pos();
        updateGL();
    }
}

void Teapots::mouseReleaseEvent(QMouseEvent * /* event */)
{
    if (rubberBandIsShown) {
        rubberBandIsShown = false;
        updateGL();
    }
}
```

鼠标事件处理器负责更新用来绘制橡皮筋选择框的 rubberBandCorner1、rubberBandCorner2 和 rubberBandIsShown 变量，并且调用 updateGL()重绘场景。重绘场景的速度很快，因为 paintGL()仅绘制一个纹理矩形以及其上的橡皮筋选择框。只有当用户改变窗口大小时，场景才在 resizeGL()中重新绘制。

以下是该应用程序的 main()函数：

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    if (!QGLFormat::hasOpenGL()) {
        std::cerr << "This system has no OpenGL support" << std::endl;
        return 1;
    }

    if (!QGLFramebufferObject::hasOpenGLFramebufferObjects()) {
        std::cerr << "This system has no framebuffer object support"
                  << std::endl;
        return 1;
    }

    Teapots teapots;
    teapots.setWindowTitle(QObject::tr("Teapots"));
    teapots.resize(400, 400);
    teapots.show();

    return app.exec();
}
```

如果系统不支持 OpenGL，或者不支持 framebuffer 对象，该函数将会发送一条错误信息，并且返回该错误代码，结束运行。

Teapots 例子告诉我们如何在离屏表面绑定纹理，以及如何使用 OpenGL 命令在该表面上绘制图形。但是我们还可以在其他方面进行有益的尝试。例如，可以使用 QPainter 代替 OpenGL 命令在 QGLFramebufferObject 或 QGLPixelBuffer 上绘制。这为在 OpenGL 场景中绘制变形字提供了一种方法。另一种常用的做法是使用 framebuffer 对象绘制场景，然后对其结果调用 toImage()生成 QImage。Qt 自带的很多例子都展示了在实际应用中的多种做法，它们都是针对 framebuffer 对象和 pbuffers 的。

第21章 创建插件

动态库(也称为共享库或者动态链接库)是存储在磁盘上一个单独文件中的独立模块,可以被多个应用程序访问。程序通常会在连接的时候指明它们所需动态库,在这种情况下,当该应用程序启动的时候,就会自动加载这些库。使用这种方式通常需要把这个库以及包含它的路径添加到应用程序的 .pro 文件中,并且需要在源程序文件中包含相应的头文件。例如:

```
LIBS        += -ldb_cxx
INCLUDEPATH += /usr/local/BerkeleyDB.4.2/include
```

另外一种方式是在需要的时候动态加载这个库,然后再解析这个库中希望使用的那些符号。Qt 提供的 QLibrary 类可以使用一种与平台无关的方式来实现这一点。只要给定一个库名字的词干(stem),QLibrary 就会在该平台的标准位置搜索这个库,并查找适当的文件。例如,给定名字 mimetype,在 Windows 上,QLibrary 就会查找 mimetype.dll;在 Linux 上,QLibrary 就会查找 mimetype.so;而在 Mac OS X 上,QLibrary 就会查找 mimetype.dylib。

利用插件,通常就可以对现存的 GUI 应用程序进行扩展。插件就是一个动态库,它为可选的额外功能提供了一个特殊接口。例如,第 5 章就创建过一个插件,它可以把自定义的窗口部件集成到 Qt 设计师中。

Qt 可以识别自己用于各域中的一套插件接口,这些域包括图像格式、数据库驱动程序、窗口部件风格、文本编解码技术以及可访问性(accessibility)等。本章的第一节将解释如何使用 Qt 插件来扩展 Qt。

为一些特殊的 Qt 应用程序创建与应用程序相关的插件也是可能的。通过 Qt 的插件框架可以很轻松地编写出这样的插件,这个框架在 QLibrary 之外添加了失效安全(crash safety)和易用性。在本章的最后两节中,将演示如何让应用程序支持插件以及如何为应用程序创建自定义插件。

21.1 利用插件扩展 Qt

可以使用很多插件类型来扩展 Qt,其中最常用的就是数据库驱动程序、图像格式、风格和文本编码解码器。对于每一种类型的插件,我们通常至少需要两个类:一个是插件封装器类,它实现了插件的通用 API 函数;另外一个是一个或多个处理器类,每个处理器类都实现了一种用于特殊类型插件的 API。通过封装器类才能访问这些处理器类。这些类如图 21.1 所示。

为了说明如何使用插件扩展 Qt,我们将在本节实现两个插件。第一个插件是非常简单的 QStyle,用于第 19 章开发的 Bronze 风格中。第二个是一个可以读取 Windows 单色光标文件的插件。

假定我们已经开发了所用的风格,那么 QStyle 插件的创建将会非常简单。我们所需要做的全部事情就是生成三个文件:一个是 .pro 文件,但这个文件与我们之前见到的都有所不同;还有一个 .h 文件和一个 .cpp 文件,由它们提供一个 QStylePlugin 子类,用作该风格的封装器。我们将从 .h 文件开始研究:

```
class BronzeStylePlugin : public QStylePlugin
{
public:
    QStringList keys() const;
    QStyle *create(const QString &key);
};
```

所有的插件至少要提供一个 keys()函数和一个 create()函数。keys()函数会返回一个该插件可以创建的对象列表。对于风格插件来说,对于字母键的大小写是不区分的,因而“mystyle”和“MyStyle”将会被看作是一样的。create()函数返回一个给定键的对象,该键必须与由 keys()函数返回的列表中的一个相同。

插件类	处理器基类
QAccessibleBridgePlugin	QAccessibleBridge
QAccessiblePlugin	QAccessibleInterface
QDecorationPlugin*	QDecoration*
QFontEnginePlugin	QAbstractFontEngine
QIconEnginePluginV2	QIconEngineV2
QImageIOPlugin	QImageIOHandler
QInputContextPlugin	QInputContext
QKbdDriverPlugin*	QWSKeyboardHandler*
QMouseDriverPlugin*	QWSMouseHandler*
QPictureFormatPlugin	N/A
QScreenDriverPlugin*	QScreen*
QScriptExtensionPlugin	N/A
QSqlDriverPlugin	QSqlDriver
QStylePlugin	QStyle
QTextCodecPlugin	QTextCodec

*仅在Linux下的Qt/Embedded中可用。

图 21.1　Qt 的插件类和处理器类

.cpp 文件几乎与 .h 文件一样简明扼要:

```
QStringList BronzeStylePlugin::keys() const
{
    return QStringList() << "Bronze";
}
```

keys()函数会返回一个由该插件提供的风格列表。这里,仅提供一种称为“Bronze”的风格:

```
QStyle *BronzeStylePlugin::create(const QString &key)
{
    if (key.toLower() == "bronze")
        return new BronzeStyle;
    return 0;
}
```

如果该键是“Bronze”(忽略大小写),就创建一个 BronzeStyle 对象并且将其返回。

在 .cpp 文件的最后,必须添加一个下面这样的宏,它可以适当地输出其风格:

```
Q_EXPORT_PLUGIN2(bronzestyleplugin, BronzeStylePlugin)
```

用于Q_EXPORT_PLUGIN2()的第一个参数项是目标库名字去除任意扩展符、前缀或者版本号之后的基本名。默认情况下,qmake 会使用当前路径名称作为基本名,这可以通过修改 .pro 文件中的 TARGET 项来替换该名字。用于Q_EXPORT_PLUGIN2()的第二个参数则是插件的类名。

插件的 .pro 文件和应用程序的 .pro 文件不同,所以我们就以查看 Bronze 风格的 .pro 文件作为结束:

```
TEMPLATE      = lib
CONFIG       += plugin
HEADERS       = ../bronze/bronzestyle.h \
                bronzestyleplugin.h
```

```
SOURCES      = ../bronze/bronzestyle.cpp \
               bronzestyleplugin.cpp
RESOURCES    = ../bronze/bronze.qrc
DESTDIR      = $$[QT_INSTALL_PLUGINS]/styles
```

默认情况下,.pro 文件使用 app 模板,但是这里必须把它指定为 lib 模板,这是因为插件只是一个库,而不是一个可单独运行的应用程序。CONFIG 行的代码是告诉 Qt,这个库并不是一个通用库,而是一个插件库。DESTDIR 指定了这个插件应当存放的目录。所有 Qt 的插件都必须放在 Qt 安装路径下 plugins 的适当子目录中。这条路径放进了 qmake 的路径,并且可以通过$$[QT_INSTALL_PLUGINS]变量获取它的取值。由于我们的插件提供了一种新的风格,所以把它放到 Qt 的 plugins/styles 子目录中。可用的目录名和插件类型可以从 http://doc.trolltech.com/4.3/plugins-howto.html 中获取。

使用 release 模式和 debug 模式为 Qt 构建的插件是不同的,因此,如果 Qt 的这两种模式都安装了,比较明智的做法是在 .pro 文件中指明要使用的是哪个文件。例如,可以添加如下一行:

```
CONFIG += release
```

一旦构建了风格,就可以准备使用了。通过在代码中指定这个风格后,应用程序就可以使用它。例如:

```
QApplication::setStyle("Bronze");
```

只需通过在运行应用程序的时候带上-style 选项,完全不必改变应用程序的源代码,也同样可以使用这一风格。例如:

```
./spreadsheet -style bronze
```

在 Qt 设计师运行时,它会自动查找各个插件。如果找到一个风格插件,它将会在它的 Form→Preview 子菜单中提供一个预览该风格的选项。

使用 Qt 插件的应用程序必须用它们打算使用的插件进行配置。Qt 插件必须放在特殊的子目录中(例如,plugins/styles 是用于自定义风格的子目录)。Qt 应用程序在可执行文件所在目录的 plugins 目录中查找插件。如果希望把 Qt 插件配置到不同与此的目录中,那么就需要在一开始就调用 QCoreApplication::addLibraryPath()来扩展插件的搜索路径,或者也可以在启动程序之前设置 QT_PLUGIN_PATH 环境变量。

我们现在已经看过简单插件了,下面将要处理的这个插件则有一定的挑战性:一个可以读取 Windows 单色光标文件(.cur 文件)的图像插件。(.cur 文件的格式如图 21.2 所示。)Windows 的光标文件可以保存分别用于表示同一光标不同大小的几幅图像。只要构建并且安装了这个光标插件,Qt 就可以读取 .cur 文件并读取出不同的光标形状(例如,通过 QImage、QImageReader 或者 QMovie),并且可以使用 Qt 的其他任意图像文件格式输出这些光标,例如 BMP、JPEG 和 PNG 格式等。

新的图像格式插件封装器必须继承 QImageIOPlugin,并且还需要重新实现一些虚函数:

```
class CursorPlugin : public QImageIOPlugin
{
public:
    QStringList keys() const;
    Capabilities capabilities(QIODevice *device,
                              const QByteArray &format) const;
    QImageIOHandler *create(QIODevice *device,
                            const QByteArray &format) const;
};
```

keys()函数返回一个该插件所能支持的图像格式的列表。capabilities()和 create()函数中的 format 参数可以认为是这个列表中的一个值。

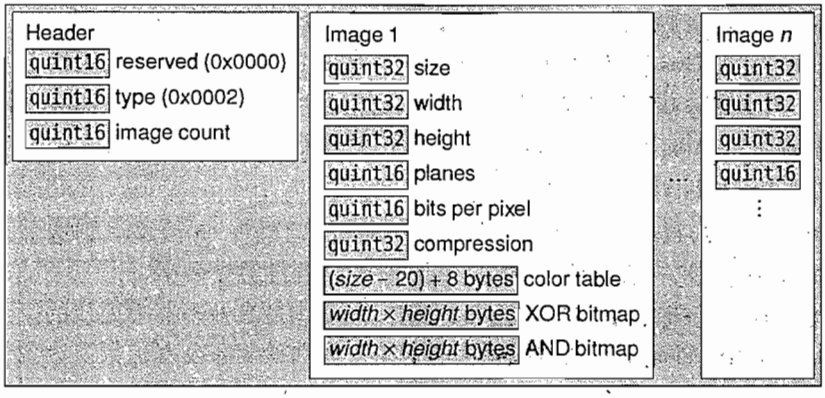


图 21.2　.cur 文件的格式

```
QStringList CursorPlugin::keys() const
{
    return QStringList() << "cur";
}
```

我们的插件仅支持一种图像格式，所以它返回一个只包含一个名字的列表。理想情况下，这个名字应当是这种格式所使用的文件扩展名。当处理带有几种不同扩展名的图像格式时（例如，JPEG 格式可以使用 .jpg 和 .jpeg 两个扩展名），我们可以为同一格式返回一个由几个元素组成的列表，而其中的每个元素都代表一种扩展名。

```
QImageIOPlugin::Capabilities
CursorPlugin::capabilities(QIODevice *device,
                           const QByteArray &format) const
{
    if (format == "cur")
        return CanRead;

    if (format.isEmpty()) {
        CursorHandler handler;
        handler.setDevice(device);
        if (handler.canRead())
            return CanRead;
    }

    return 0;
}
```

函数 capabilities()返回该图像处理器利用给定的图像格式所具有的处理能力。一共有三种处理能力（CanRead、CanWrite 和 CanReadIncremental），并且返回值是所能应用的那些能力的按位“或”后的结果。

如果格式是“cur”，我们的实现就会返回 CanRead。如果没有给定格式，就创建一个光标处理器并且检查它是否具有从给定设备读取数据的能力。canRead()函数只是粗略查看一下数据，判断它是否可以识别这个文件，但并不会改变文件指针。如果能力值是 0，则意味着这个处理器不能读取或者写入该个文件。

```
QImageIOHandler *CursorPlugin::create(QIODevice *device,
                                      const QByteArray &format) const
{
    CursorHandler *handler = new CursorHandler;
    handler->setDevice(device);
    handler->setFormat(format);
    return handler;
}
```

在打开光标文件时(例如,通过 QImageReader),就会利用设备指针和作为格式的"cur"为参数,调用这个插件封装器的 create()函数。我们创建一个 CursorHandler 实例并且用指定的设备和格式对其进行设置。调用者会获得这个处理器的所有权并且会在不需要时将其删除。如果需要读取多个文件,那么就需要为每个文件各自创建一个新的处理器。

```
Q_EXPORT_PLUGIN2(cursorplugin, CursorPlugin)
```

在 .cpp 文件的最后,我们使用 Q_EXPORT_PLUGIN2()宏确保 Qt 能够识别这个插件。第一个参数是我们希望给这个插件起的任意名字,第二个参数是插件类的名字。

对 QImageIOPlugin 的继承是很简单的。插件的实际工作是在处理器中完成的。图像格式处理器必须继承 QImageIOHandler,并且需要重新实现它的一些或者全部公有函数。让我们先从头文件开始看起:

```
class CursorHandler : public QImageIOHandler
{
public:
    CursorHandler();

    bool canRead() const;
    bool read(QImage *image);
    bool jumpToNextImage();
    int currentImageNumber() const;
    int imageCount() const;

private:
    enum State { BeforeHeader, BeforeImage, AfterLastImage, Error };

    void readHeaderIfNecessary() const;
    QBitArray readBitmap(int width, int height, QDataStream &in) const;
    void enterErrorState() const;

    mutable State state;
    mutable int currentImageNo;
    mutable int numImages;
};
```

所有公有函数的签名都是固定的。作为一个只读处理器,像 write()这样的一些特殊函数我们并没有必要去重新实现它们,所以就将它们省略了。成员变量都使用了 mutable 关键字,因为它们会在 const 函数中得到修改。

```
CursorHandler::CursorHandler()
{
    state = BeforeHeader;
    currentImageNo = 0;
    numImages = 0;
}
```

在构造处理器的时候,先从设置它的状态开始。我们把当前的光标图像设置成第一个光标,其编号为 1。但由于把 numImages 设置成了 0,显然是说我们还没有任何光标图像。

```
bool CursorHandler::canRead() const
{
    if (state == BeforeHeader) {
        return device()->peek(4) == QByteArray("\0\0\2\0", 4);
    } else {
        return state != Error;
    }
}
```

每当处理器在判断是否能从这个设备中读取更多数据的时候,就会调用 canRead()函数。如果在读取任意数据之前已经调用了这个函数,但还处于 BeforeHeader 状态,那么就需要检查用以识别 Windows 光标文件的那个特殊签名了。这里的 QIODevice::peek()调用可以用来读取设备文件的

前 4 个字节，但并不会改变该设备的文件指针。如果此后再调用 canRead()，只要没有发生错误，就一直返回 true。

```
int CursorHandler::currentImageNumber() const
{
    return currentImageNo;
}
```

这个简单函数返回设备文件指针指向的光标的编号。

处理器一旦构造完毕，就可以按任意顺序来调用它的任一公有函数了。这就存在一个潜在问题，由于我们必须假定只能连续读取，所以在做其他任何事情之前，都需要读取一次文件头。通过在那些依赖头文件并已经读取了头文件的函数中调用 readHeaderIfNecessary()函数，就可以解决这一问题。

```
int CursorHandler::imageCount() const
{
    readHeaderIfNecessary();
    return numImages;
}
```

这个函数返回在这个文件中图像的个数。对于一个不曾发生过读取错误的有效文件，它会返回一个至少是 1 的计数值。

下一个函数有些麻烦，所以将一段一段地查看它：

```
bool CursorHandler::read(QImage *image)
{
    readHeaderIfNecessary();

    if (state != BeforeImage)
        return false;
```

read()函数会读取从当前设备文件指针位置处开始的任何一个图像的数据。如果文件头读取顺利，或者是在一个已经读取完毕的图像之后，并且该设备指针又恰好在一个图像的开始处，那么就可以读取下一个图像了。

```
    quint32 size;
    quint32 width;
    quint32 height;
    quint16 numPlanes;
    quint16 bitsPerPixel;
    quint32 compression;

    QDataStream in(device());
    in.setByteOrder(QDataStream::LittleEndian);
    in >> size;
    if (size != 40) {
        enterErrorState();
        return false;
    }

    in >> width >> height >> numPlanes >> bitsPerPixel >> compression;
    height /= 2;

    if (numPlanes != 1 || bitsPerPixel != 1 || compression != 0) {
        enterErrorState();
        return false;
    }

    in.skipRawData((size - 20) + 8);
```

我们创建一个 QDataStream 读取这个设备。必须把字节顺序设置成与 .cur 文件格式规范相匹配的顺序。由于整数和浮点数的格式在数据流的不同版本之间不存在变化，所以就没有必要设置 QDataStream 的版本号。接下来，读入光标文件头部数据中的各项，并且跳过与文件头不相关的项和使用 QDataStream::skipRawData()函数的 8 字节颜色表。

我们必须考虑这个格式的所有特性——例如,把高度减半是因为 .cur 格式所给的高度是实际图像高度的两倍。在单色的 .cur 文件中,bitsPerPixel 和 compression 的值总是分别为 1 和 0。如果遇到任何问题,就可以调用 enterErrorState()并且返回 false。

```
QBitArray xorBitmap = readBitmap(width, height, in);
QBitArray andBitmap = readBitmap(width, height, in);

if (in.status() != QDataStream::Ok) {
    enterErrorState();
    return false;
}
```

文件中接下来的两项是两个位图,一个是 XOR 掩码,另外一个是 AND 掩码。我们把这两项读入到 QBitArray 中,而不是 QBitmap 中。QBitmap 是一个类,设计用于在屏幕上画线和绘图,但是在这里,我们需要的只是一个普通位数组。

当读取这个文件之后,就检查 QDataStream 的状态。之所以这样做,是因为如果 QDataStream 中的状态是一个错误状态,那么它就会保持这种错误状态并且一直返回 0 值。例如,如果对位数组的第一位读取失败,那么对第二位的尝试读取将产生一个空的 QBitArray。

```
*image = QImage(width, height, QImage::Format_ARGB32);

for (int i = 0; i < int(height); ++i) {
    for (int j = 0; j < int(width); ++j) {
        QRgb color;
        int bit = (i * width) + j;

        if (andBitmap.testBit(bit)) {
            if (xorBitmap.testBit(bit)) {
                color = 0x7F7F7F7F;
            } else {
                color = 0x00FFFFFF;
            }
        } else {
            if (xorBitmap.testBit(bit)) {
                color = 0xFFFFFFFF;
            } else {
                color = 0xFF000000;
            }
        }
        image->setPixel(j, i, color);
    }
}
```

我们以正确的大小构造一个新的 QImage 并且将其赋值给 * image。然后,遍历 XOR 和 AND 两个位数组中的每一个像素并且把它们转换为 32 位 AGRB 的颜色格式。如下表所示,XOR 和 AND 位数组可以用于获取每个光标像素的颜色值:

AND	XOR	结果
1	1	背景像素颜色反色
1	0	透明像素
0	1	白色
0	0	黑色

要获取黑色、白色和透明像素不存在什么问题,但是在不知道背景像素原始颜色的情况下,我们却没有办法使用 ARGB 颜色格式获得背景像素颜色反色的像素。作为背景像素原始颜色的替代品,我们使用一种半透明的灰颜色(0x7F7F7F7F)。

```
    ++currentImageNo;
    if (currentImageNo == numImages)
        state = AfterLastImage;
    return true;
}
```

一旦完成了这个图像的读取，就可以更新当前图像编号。如果读取的是最后一个图像，则同时还需要更新 state 的值。此时，设备将指向下一个图像或者文件的最后位置处。

```
bool CursorHandler::jumpToNextImage()
{
    QImage image;
    return read(&image);
}
```

这个 jumpToNextImage()函数用于跳过一个图像。为简便起见，我们只简单调用 read()并忽略作为结果的 QImage。一种更为有效的实现应当是使用保存在 .cur 文件头中的信息并且直接跳到这个文件中的适当偏移量处。

```
void CursorHandler::readHeaderIfNecessary() const
{
    if (state != BeforeHeader)
        return;

    quint16 reserved;
    quint16 type;
    quint16 count;

    QDataStream in(device());
    in.setByteOrder(QDataStream::LittleEndian);

    in >> reserved >> type >> count;
    in.skipRawData(16 * count);

    if (in.status() != QDataStream::Ok || reserved != 0
            || type != 2 || count == 0) {
        enterErrorState();
        return;
    }
    state = BeforeImage;
    currentImageNo = 0;
    numImages = int(count);
}
```

私有函数 readHeaderIfNecessary()会被 imageCount()和 read()调用。如果该文件的文件头已经被读取过，那么 state 就不会再是 BeforeHeader 了，所以可以立即返回。否则，就在这个设备上开启一个数据流，读入一些普通数据(包括这个文件中的光标个数)，并且把 state 设置为 BeforeImage。最后，把这个设备的文件指针指向第一个图像的前面。

```
void CursorHandler::enterErrorState() const
{
    state = Error;
    currentImageNo = 0;
    numImages = 0;
}
```

如果有错误发生，就认为不存在有效图像并且把 state 设置为 Error。一旦进入 Error 状态，这个处理器的状态就不能再改变了。

```
QBitArray CursorHandler::readBitmap(int width, int height,
                                    QDataStream &in) const
{
    QBitArray bitmap(width * height);
    quint32 word = 0;
    quint8 byte;
```

```
    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            if ((j % 32) == 0) {
                word = 0;
                for (int k = 0; k < 4; ++k) {
                    in >> byte;
                    word = (word << 8) | byte;
                }
            }

            bitmap.setBit(((height - i - 1) * width) + j,
                          word & 0x80000000);
            word <<= 1;
        }
    }
    return bitmap;
}
```

readBitmap()函数用于读取一个光标的 AND 和 XOR 掩码。这些掩码具有两种不同寻常的特点。第一,它们按照从下到上的顺序保存行信息,而不是更常用的从上到下的方法。第二,数据的字节序列看起来应当与 .cur 文件中其他地方的字节序列相反。考虑到这一点,我们必须在setBit()调用中倒置 y 坐标,并且在读取掩码值的时候,每次只读取一个字节,然后再利用位移位和掩码提取它们的正确数值。

要构建这个插件,我们必须使用一个 .pro 文件,它与之前的 Bronze 风格插件中用过的 .pro 文件很相似:

```
TEMPLATE      = lib
CONFIG       += plugin
HEADERS       = cursorhandler.h \
                cursorplugin.h
SOURCES       = cursorhandler.cpp \
                cursorplugin.cpp
DESTDIR       = $$[QT_INSTALL_PLUGINS]/imageformats
```

这样,我们就完成了 Windows 光标插件。对于其他图像格式的插件,应当遵循同样的模式,虽然可能还需要实现更多的 QImageIOHandler API,特别是用于输出图像的那些函数。其他类型的插件也具有同样的模式,即使用一个插件封装器导出一个或者多个提供底层功能的处理器。

21.2 使应用程序感知插件

一个应用程序插件就是实现了一个或多个接口(interface)的动态库。接口就是由专有的纯虚函数组成的类。应用程序和插件之间的通信是通过接口的虚表(virtual table)来完成的。在这一节中,我们将主要集中看看如何在一个 Qt 应用程序中通过它的接口使用插件。在下一节,将说明如何实现插件。

为了给出一个具体的例子,我们将会创建一个如图 21.3 所示的简单的 Text Art 应用程序。这些效果是通过一些插件提供给文本的,应用程序可以获得由每个插件提供的文本效果构成的列表并且遍历它们,然后在 QListWidget 中逐项显示这些效果。

Text Art 应用程序定义了一个接口:

```
class TextArtInterface
{
public:
    virtual ~TextArtInterface() { }

    virtual QStringList effects() const = 0;
    virtual QPixmap applyEffect(const QString &effect,
                                const QString &text,
```

```
                                const QFont &font, const QSize &size,
                                const QPen &pen,
                                const QBrush &brush) = 0;
};

Q_DECLARE_INTERFACE(TextArtInterface,
                    "com.software-inc.TextArt.TextArtInterface/1.0")
```

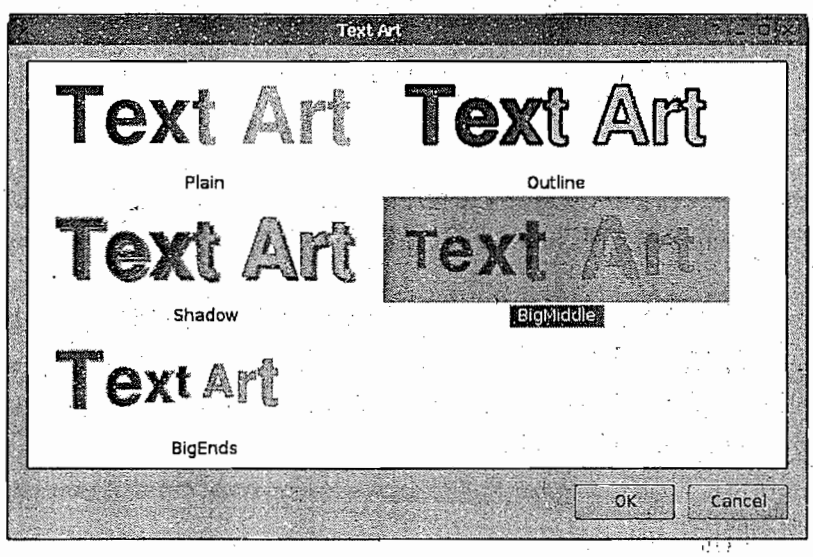


图 21.3　Text Art 应用程序

接口类通常都需要声明一个虚析构函数，一个返回 QStringList 的虚函数和一个或者多个其他虚函数。这个析构函数主要是为了抑制(silence)编译器，因为如果不这样做，编译器就会抱怨一个具有虚函数的类却没有一个虚析构函数。在这个例子中，effects()函数可以返回该插件能够提供的文本效果的列表。可以把这个列表看成一个键列表。每次调用其他几个函数中的一个函数时，就需要把这些键中的一个作为第一个参数进行传递，这样才有可能在一个插件中实现多个效果。

最后，我们使用 Q_DECLARE_INTERFACE()宏使这个接口与一个标识符关联起来。这个标识符通常包含 4 个部分：一个倒置了的域名，用来说明这个接口的创建者是谁；一个应用程序的名字；一个接口的名字；还有一个版本号。只要我们改变了这个接口(例如，增加了一个虚函数或者改变了一个已有函数的签名)，就必须记得要增加版本号；否则，这个应用程序可能会因为试图访问一个过期的插件而崩溃。

在一个称为 TextArtDialog 的类中，实现了这个应用程序。我们将只说明能够让这个应用程序感知插件相关的代码。先从构造函数开始：

```
TextArtDialog::TextArtDialog(const QString &text, QWidget *parent)
    : QDialog(parent)
{
    listWidget = new QListWidget;
    listWidget->setViewMode(QListWidget::IconMode);
    listWidget->setMovement(QListWidget::Static);
    listWidget->setIconSize(QSize(260, 80));
    ...
    loadPlugins();
    populateListWidget(text);
    ...
}
```

这个构造函数创建了一个 QListWidget 列出所有可用的效果。它调用 loadPlugins()私有函数查

找并且加载实现了 TextArtInterface 的每一个插件,并且通过调用另外一个私有函数 populateListWidget()弹出这个列表窗口部件。

```
void TextArtDialog::loadPlugins()
{
    QDir pluginsDir = directoryOf("plugins");
    foreach (QString fileName, pluginsDir.entryList(QDir::Files)) {
        QPluginLoader loader(pluginsDir.absoluteFilePath(fileName));
        if (TextArtInterface *interface =
                    qobject_cast<TextArtInterface *>(loader.instance()))
            interfaces.append(interface);
    }
}
```

在 loadPlugins()中,我们先从搜索该应用程序的 plugins 目录开始。然后,试图加载在这个应用程序的 plugins 目录中的所有文件。directoryOf()函数与在第 17 章中使用的 directoryOf()函数是同一个函数。

如果我们试图加载的文件是一个和这个应用程序的 Qt 版本号相同的 Qt 插件,那么 QPluginLoader::instance()将会返回一个指向这个 Qt 插件的 QObject * 。我们使用 qobject_cast <T>()检查这个插件是否实现了 TextArtInterface。每次强制转换成功的时候,就把这个接口添加到 TextArtDialog 的接口列表(类型为 QList <TextArtInterface *>)中。

一些应用程序也许想加载两种或者更多种不同的接口。在这种情况下,获得接口的代码看起将更像是如下所示的样子:

```
QObject *plugin = loader.instance();
if (TextArtInterface *i = qobject_cast<TextArtInterface *>(plugin))
    textArtInterfaces.append(i);
if (BorderArtInterface *i = qobject_cast<BorderArtInterface *>(plugin))
    borderArtInterfaces.append(i);
if (TextureInterface *i = qobject_cast<TextureInterface *>(plugin))
    textureInterfaces.append(i);
```

如果使用多重继承方式,那么同一个插件就可能成功强制转换出超过一个的接口指针。

```
void TextArtDialog::populateListWidget(const QString &text)
{
    QFont font("Helvetica", iconSize.height(), QFont::Bold);
    QSize iconSize = listWidget->iconSize();
    QPen pen(QColor("darkseagreen"));

    QLinearGradient gradient(0, 0, iconSize.width() / 2,
                             iconSize.height() / 2);
    gradient.setColorAt(0.0, QColor("darkolivegreen"));
    gradient.setColorAt(0.8, QColor("darkgreen"));
    gradient.setColorAt(1.0, QColor("lightgreen"));

    foreach (TextArtInterface *interface, interfaces) {
        foreach (QString effect, interface->effects()) {
            QListWidgetItem *item = new QListWidgetItem(effect,
                                                        listWidget);
            QPixmap pixmap = interface->applyEffect(effect, text, font,
                                                    iconSize, pen,
                                                    gradient);
            item->setData(Qt::DecorationRole, pixmap);
        }
    }
    listWidget->setCurrentRow(0);
}
```

在 populateListWidget()函数的一开始,就创建了一些要传递给 applyEffect()函数的变量,特别是创建了一种字体、一支画笔和一个线性渐变器(gradient)。然后,它遍历由 loadPlugins()发现的每一

个TextArtInterface。对于每一个接口所提供的每一种效果，会使用它的效果名称的文本和使用applyEffect()生成的QPixmap来创建一个新的QListWidgetItem。

在这一节中，我们已经看到在构造函数中是如何调用loadPlugins()加载插件的，并且看到了在populateListWidget()中是如何使用它们的。不论是否有插件提供TextArtInterface、或者只提供一个、或者提供多个，这段代码都可以很好地处理这些情况。另外，还可以在之后的时间里添加一些额外的插件：每当应用程序启动的时候，它都会加载所发现的能够提供它想要的接口的那些插件。这样就可以很容易地扩展应用程序的功能，而不需要改变应用程序本身。

21.3 编写应用程序的插件

一个应用程序的插件就是QObject和它想要提供的接口的一个子类。与本书一起使用的例子中包含了两个插件，它们可用于上一节中讲到的Text Art应用程序，这样我们就可以说明该应用程序可以正确地处理多个插件。

这里，我们仅查看其中一个的代码：Basic Effects插件。假定这个插件的代码是放在一个称为basiceffectsplugin的目录中，并且Text Art应用程序是放在和它并列的一个称为textart的目录中。这里是该插件类的声明：

```
class BasicEffectsPlugin : public QObject, public TextArtInterface
{
    Q_OBJECT
    Q_INTERFACES(TextArtInterface)

public:
    QStringList effects() const;
    QPixmap applyEffect(const QString &effect, const QString &text,
                        const QFont &font, const QSize &size,
                        const QPen &pen, const QBrush &brush);
};
```

这个插件只实现了一个接口：TextArtInterface。除了Q_OBJECT之外，还必须为继承的每一个接口使用Q_INTERFACES()宏，以确保moc和qobject_cast <T>()可以一起正常工作。

```
QStringList BasicEffectsPlugin::effects() const
{
    return QStringList() << "Plain" << "Outline" << "Shadow";
}
```

effects()函数返回插件所支持的文本效果的列表。这个插件支持三种效果，所以返回了一个包含每个效果名称的列表。

applyEffect()函数提供了插件的功能并且它稍微有些复杂，所以我们将分段来查看它：

```
QPixmap BasicEffectsPlugin::applyEffect(const QString &effect,
        const QString &text, const QFont &font, const QSize &size,
        const QPen &pen, const QBrush &brush)
{
    QFont myFont = font;
    QFontMetrics metrics(myFont);
    while ((metrics.width(text) > size.width()
            || metrics.height() > size.height())
           && myFont.pointSize() > 9) {
        myFont.setPointSize(myFont.pointSize() - 1);
        metrics = QFontMetrics(myFont);
    }
```

我们想确保给定的文本能够尽可能地占满所给定的大小。基于这个原因，我们使用字体规格(metrics)看看文本是不是太大了，并且如果是这样的话，就进入到一个减小字号的循环当中，直到

找到一个合适的字号,或者直到达到 9 磅(point)的字号大小时为止,这也是我们设定的最小值。

```
QPixmap pixmap(size);

QPainter painter(&pixmap);
painter.setFont(myFont);
painter.setPen(pen);
painter.setBrush(brush);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setRenderHint(QPainter::TextAntialiasing, true);
painter.setRenderHint(QPainter::SmoothPixmapTransform, true);
painter.eraseRect(pixmap.rect());
```

我们根据所需大小创建了一个像素映射(pixmap),和一个要在它上面进行绘制的绘图器(painter)。还设置了一些着色提示,用来保证最可能实现的平滑结果。eraseRect()的调用则会使用背景色清空这个像素映射。

```
if (effect == "Plain") {
    painter.setPen(Qt::NoPen);
} else if (effect == "Outline") {
    QPen pen(Qt::black);
    pen.setWidthF(2.5);
    painter.setPen(pen);
} else if (effect == "Shadow") {
    QPainterPath path;
    painter.setBrush(Qt::darkGray);
    path.addText(((size.width() - metrics.width(text)) / 2) + 3,
                 (size.height() - metrics.descent()) + 3, myFont,
                 text);
    painter.drawPath(path);
    painter.setBrush(brush);
}
```

对于“Plain”效果,就不需要任何轮廓(outline)了。对于“Outline”效果,会忽略初始的画笔,并且会创建一个黑色的、2.5 像素宽的画笔。对于“Shadow”效果,需要先绘制阴影,以便能够使文本绘制在“Shadow”之上。

```
    QPainterPath path;
    path.addText((size.width() - metrics.width(text)) / 2,
                 size.height() - metrics.descent(), myFont, text);
    painter.drawPath(path);

    return pixmap;
}
```

现在已经为每一个文本效果设置好了画笔和画刷,并且在“Shadow”效果中已经画好了阴影。这样就为渲染文本做好了准备。文本是水平居中的,并且在这个像素映射的底部留有足够的空间来绘制后面的部分。

```
Q_EXPORT_PLUGIN2(basiceffectsplugin, BasicEffectsPlugin)
```

在 .cpp 文件的最后,我们使用 Q_EXPORT_PLUGIN2()宏来让插件变得可以被 Qt 使用。

.pro 文件和本章前面的 Bronze 风格插件中的 .pro 文件类似:

```
TEMPLATE      = lib
CONFIG       += plugin
HEADERS       = ../textart/textartinterface.h \
                basiceffectsplugin.h
SOURCES       = basiceffectsplugin.cpp
DESTDIR       = ../textart/plugins
```

如果本章已经能够让你对应用程序的插件感到兴奋的话,那么你也许会希望进一步研究一下 Qt 提供的更高级的 Plug & Paint 例子。这个应用程序支持三种不同的接口,并且包含了一个有用的 Plugin Information(插件信息)对话框,其中列出了所有可用于这个应用程序的插件和接口。

第 22 章　应用程序脚本

脚本是采用解释型语言写成的程序，增强了系统的灵活性。有些脚本就是独立的应用程序，其他的则嵌入到应用程序中运行。从 Qt 4.3 开始包含了 QtScript，这是一个可以使用 ECMAScript 的模块；ECMAScript 是标准版本的 JavaScript。这个模块是早前 Trolltech 产品 Qt 应用程序脚本(QSA，Qt Script for Applications)的全面改写，可以提供对 ECMAScript 版本 3 的全面支持。

ECMAScript 是一个由 Ecma 国际组织标准化的官方语言名称。它构成了 JavaScript(Mozilla)、JScript(Microsoft)和 ActionScript(Adobe)的基础。尽管该语言的语法表面上与 C++ 和 Java 语言相类似，但潜在的概念是根本不同的，从而使它独立于许多其他的面向对象的编程语言。在本章的第一节，我们将快速了解 ECMAScript 语言，介绍怎样在 Qt 中运行 ECMAScript 代码。如果你已经了解了 JavaScript，或者其他基于 ECMAScript 的语言，那么就可以跳过这一节。

第二节将介绍如何在 Qt 应用程序中添加脚本支持。这使得用户可以在应用程序已提供功能的基础上添加他们自己的功能。这一方法也常常被用来支持用户，技术支持人员可以以脚本的形式提供缺陷修正和解决方案。

第三节将介绍如何结合 ECMAScript 代码和 Qt 设计师创建的窗体开发 GUI 前端。这一方法对那些不喜欢 C++ 开发中的“编译、链接、运行”周期循环，而更喜欢脚本方法的开发者很有吸引力。它可以使那些有 JavaScript 经验的用户不用学习 C++ 就能够设计功能齐全的用户界面。

在最后一节，将介绍如何开发依赖于 C++ 组件的脚本。这可以通过任意的 C++ 和++ /Qt 组件完成，不用构思就可以设计出来。这个方法很有用，特别是当几个程序需要使用相同的基本组件编写程序，或者是当我们想让非 C++ 程序员使用 C++ 功能的时候更是如此。

22.1　ECMAScript 语言概述

这一节对 ECMAScript 语言作一简单的介绍，以便更好地理解本章的代码片段，并且可以开始编写你自己的脚本。Mozilla 基金会的网站上保存有更完整的用户指南 http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide，此外，David Flanagan 的“JavaScript: The Definitive Guide“(O'Reilly, 2006)也是值得推荐的，它既可以作为指南又可以当作参考文档。官方的 ECMAScript 规范可以查看在线文档 http://www.ecmainternational.org/publications/standards/Ecma-262.htm。

基本的 ECMAScript 控制体——if 语句、for 循环以及 while 循环等，这些都与 C++ 和 Java 是相同的。ECMAScript 也提供或多或少的相同的赋值、关系和算术运算符。ECMAScript 字符串支持使用 + 的串联形式和 += 的叠加形式。

为了掌握 ECMAScript 的语法，我们首先来学习下面一个程序，它可以打印 1000 以下所有素数的列表：

```
const MAX = 1000;
var isPrime = new Array(MAX);

for (var i = 2; i < MAX; ++i)
    isPrime[i] = true;

for (var i = 2; i < MAX; ++i) {
```

```
    if (isPrime[i]) {
        for (var j = i; i * j < MAX; ++j)
            isPrime[i * j] = false;
    }
}

for (var i = 2; i < MAX; ++i) {
    if (isPrime[i])
        print(i);
}
```

从一个 C++ 程序员的角度来看,可能 ECMAScript 最显著的特点是变量而不是显式类型,利用 var 关键字就可以定义一个变量。只读变量可以使用 const 来声明,而不是使用 var 来声明。前面程序中,另外一个值得注意的特点是没有 main()函数。函数外面的代码会被立即执行,即从文件的开头一直执行到文件的末尾。

与 C++ 不同,语句末尾的分号在 ECMAScript 中通常是可选的。利用改进的规则,解释器能够自动添加大部分遗漏的分号。尽管如此,还是要尽量自己输入分号,这样可以避免意外的发生。

运行上面的程序,我们可以使用 qscript 解释器,它位于 Qt 的 examples/script/qscript 目录下。如果解释器被调用时带有一个作为参数的文件名,那么这个文件就会作为 ECMAScript 而得以执行;否则,就开始一个交互式会话。

如果声明 var 变量时没有提供初始值,那么默认值就是 undefined,这是一个 Undefined 型的特殊值。稍后可以使用赋值操作符(=)把它指定为任何类型的任意值。请看下面的例子:

```
var x;
typeof x;           // returns "undefined"

x = null;
typeof x;           // returns "null"

x = true;
typeof x;           // returns "boolean"

x = 5;
typeof x;           // returns "number"

x = "Hello";
typeof x;           // returns "string"
```

typeof 操作符返回一个小写的字符串,它返回与保存在变量中的值相关的数据类型。ECMAScript 定义了 5 种基本的数据类型:Undefined、Null、Boolean、Number 和 String。Undefined 和 Null 类型是 undefined 和 null 常量的特殊类型。Boolean 类型包含两个值:true 和 false。Number 类型用来保存浮点数。String 类型则可以保存 Unicode 字符串。

变量也可以保存对象和函数,分别对应于 Object 和 Function 数据类型。例如:

```
x = new Array(10);
typeof x;           // returns "object"

x = print;
typeof x;           // returns "function"
```

如同 Java 一样,ECMAScript 也会区分基本类型和对象类型。基本类型类似 C++ 的值类型,例如 int 和 QString。这些不用 new 操作符创建,可以按值复制。相反,创建对象类型则必须使用 new 操作符,这些类型的变量只保存对象的引用(指针)。当使用 new 创建对象时,我们不用担心释放内存,因为垃圾信息收集器会自动处理这件事。

如果使用 var 关键字为一个变量赋值而没有提前声明它,那么该变量将被当作全局变量。如果试图读取一个不存在的变量的值,则会得到一个 ReferenceError 异常。可以使用 try...catch 语句捕获这个异常,如下所示:

```
try {
    print(y);
} catch (e) {
    print(e.name + ": " + e.message);
}
```

如果变量 y 确实不存在，那么信息“ReferenceError:y is not defined”就会被输出到控制台上。

如果未定义的变量会在程序中造成危害的话，那么这些已定义但却保存 undefined 常量的变量也会如此——在使用 var 声明一个变量时，如果没有提供初始值就会以此作为默认值。要测试 undefined，我们可以使用严格的比较操作符 === 或者 !==。例如：

```
var x;
...
var y = 0;
if (x !== undefined)
    y = x;
```

类似的 == 和 != 比较操作符在 ECMAScript 中也可以使用，但不像 === 和 !==，当被比较的值有不同的类型时，它们通常返回 true。例如，24 == “24”，以及 null == undefined 都可以返回 true，而 24 === “24”和 null === undefined 返回 false。

现在看一个更复杂些的程序，介绍在 ECMAScript 中如何定义自己的函数：

```
function square(x)
{
    return x * x;
}

function sumOfSquares(array)
{
    var result = 0;
    for (var i = 0; i < array.length; ++i)
        result += square(array[i]);
    return result;
}

var array = new Array(100);
for (var i = 0; i < array.length; ++i)
    array[i] = (i * 257) % 101;

print(sumOfSquares(array));
```

函数用 function 关键字定义。为了保持 ECMAScript 的动态性，参数声明没有类型，函数没有显式的返回类型。

通过查看代码，我们可以猜测 square()调用的参数是 Number，sumOfSquares()调用的参数是 Array 对象，但也不一定就是这样。例如，square(“7”)将返回 49，因为 ECMAScript 的乘法操作会在数字上下文中把字符串转换为数字。同样，sumOfSquare()函数不仅可以使用 Array 对象，也可以使用具有相同接口的其他对象。

一般地，ECMAScript 会使用鸭类鉴别原理(duck typing principle)：“如果它走路像鸭子，叫声像鸭子，那它一定就是鸭子”。这与 C++ 和 Java 的强类型鉴别相对立，强类型鉴别的参数类型必须被定义，变量必须与声明的类型相匹配。

在前面的例子中，sumOfSquares()以硬编码的方式为数组的每个元素计算 square()。我们可以使它更灵活，让它接收一个一元函数作为第二个参数，并把它改名为 sum()。

```
function sum(array, unaryFunc)
{
    var result = 0;
    for (var i = 0; i < array.length; ++i)
        result += unaryFunc(array[i]);
    return result;
```

```
}

var array = new Array(100);
for (var i = 0; i < array.length; ++i)
    array[i] = (i * 257) % 101;

print(sum(array, square));
```

调用 sum(array,square)跟调用 sumOfSquares(array)一样。除了定义 square()函数外,还可以传递一个匿名函数给 sum():

```
print(sum(array, function(x) { return x * x; }));
```

除了定义一个 array 变量,也可以传递一个数组字面值:

```
print(sum([4, 8, 11, 15], function(x) { return x * x; }));
```

ECMAScript 允许向函数提供比定义的参数更多的参数。额外的参数可以通过 arguments 数组访问。请看下面的例子:

```
function sum(unaryFunc)
{
    var result = 0;
    for (var i = 1; i < arguments.length; ++i)
        result += unaryFunc(arguments[i]);
    return result;
}

print(sum(square, 1, 2, 3, 4, 5, 6));
```

这里,sum()函数被定义为带可变参数。第一个参数是我们想使用的函数,其他的参数是需要求和的数字。当遍历 arguments 数组时,必须跳过位置 0 这一项,因为它对应的是 unaryFunc 一元函数。也可以从参数列表中忽略 unaryFunc 参数,把它从 arguments 数组中提取出来:

```
function sum()
{
    var unaryFunc = arguments[0];
    var result = 0;
    for (var i = 1; i < arguments.length; ++i)
        result += unaryFunc(arguments[i]);
    return result;
}
```

arguments 数组可以根据参数类型或者它们的数量重载函数的功能。例如,假设我们想让 sum()带有一个可选的一元参数,以及一个数字的列表,并且允许我们这样调用它:

```
print(sum(1, 2, 3, 4, 5, 6));
print(sum(square, 1, 2, 3, 4, 5, 6));
```

为了达到这个目的,可以这样实现 sum()函数:

```
function sum()
{
    var unaryFunc = function(x) { return x; };
    var i = 0;

    if (typeof arguments[0] == "function") {
        unaryFunc = arguments[0];
        i = 1;
    }

    var result = 0;
    while (i < arguments.length)
        result += unaryFunc(arguments[i++]);
    return result;
}
```

如果第一个参数是一个函数,在求和之前,这个函数会处理每个数字;否则,可以使用恒等函数 function(){return x;}。

对于 C++ 程序员,认为 ECMAScript 最难的就是它的对象模型。ECMAScript 是一种基于对象(object-based)的面向对象的语言,以使它区别于 C++、C#、Java、Simula 和 Smalltalk,它们都是基于类的。代替类的概念,ECMAScript 提供了一种更低层次的机制。

允许我们在 ECMAScript 中实现类的第一个机制是构造函数。构造函数是一个能被 new 操作符调用的函数。例如,以下就是 Shape 对象的构造函数:

```
function Shape(x, y) {
    this.x = x;
    this.y = y;
}
```

Shape 构造函数有两个参数,它基于传递给它的值去初始化新对象的 x 属性和 y 属性(成员变量)。this 关键字指被创建的对象。在 ECMAScript 中,对象本质上是一个属性集,属性随时可以被添加、删除或者修改。属性会在第一次进行设置的时候得到创建,因此当我们在构造函数中为 this.x 和 this.y 赋值时,x 和 y 的属性就被创建了。

对于 C++ 和 Java 开发者来说,一个普遍的错误就是在访问对象属性时忘记使用 this 关键字。在前面的例子中,这似乎不太可能,因为语句 x = x 看起来很可疑,但在其他的例子中,这将导致创建一个虚假的全局变量。

为了创建 Shape 对象的实例,可以这样使用 new 操作符:

```
var shape = new Shape(10, 20);
```

如果在 shape 变量上使用 typeof 操作符,则数据类型是 Object,而不是 Shape。如果想确定 Shape 构造函数是否创建了对象,可以使用 instanceof 操作符:

```
var array = new Array(100);
array instanceof Shape;              // returns false

var shape = new Shape(10, 20);
shape instanceof Shape;              // returns true
```

ECMAScript 允许任何函数作为构造函数。尽管如此,如果函数没有对 this 对象做任何修改,作为构造函数调用它也没有意义。相反,构造函数也可以作为普通函数而得到调用,但这也并没有什么意义。

除了基本数据类型,ECMAScript 提供内置的构造函数,允许我们实例化基本对象类型,比如 Array、Date 和 RegExp。与基本数据类型相对应的其他构造函数,允许我们创建能够保存原始值的对象。valueOf()成员函数可以用来查询保存在对象里的原始值。例如:

```
var boolObj = new Boolean(true);
typeof boolObj;                      // returns "object"

var boolVal = boolObj.valueOf();
typeof boolVal;                      // returns "boolean"
```

图 22.1 列举了 ECMAScript 提供的内置的全局常量、函数和对象。下一节将在内置的功能中添加额外的、应用程序相关的 C++ 组件。

前面已经了解了如何在 ECMAScript 中定义构造函数,以及如何向构造的对象中添加成员变量。通常,我们也需要定义成员函数。因为在 ECMAScript 中,函数被看作是一等公民,这就变得异常容易。这里有一个 Shape 构造函数的新版本,它带有两个成员函数,manhattanPos()和 translate():

```
function Shape(x, y) {
    this.x = x;
    this.y = y;
```

```
    this.manhattanPos = function() {
        return Math.abs(this.x) + Math.abs(this.y);
    };
    this.translate = function(dx, dy) {
        this.x += dx;
        this.y += dy;
    };
}
```

常量	
NaN	IEEE 754 非数值
Infinity	正无穷大(+∞)
Undefined	未初始化变量的默认值

函数	
print(x)[①]	在控制台打印一个值
eval(str)	执行一个 ECMAScript 程序
parseInt(str,base)	把字符串转换为整型
parseFloat(str)	把字符串转换为浮点型
isNaN(n)	如果 n 是 NaN,返回 true
isFinite(n)	如果 n 是非 NaN、+∞或 -∞的数字,返回 true
decodeURI(str)	把 8 位编码 URI 转换为 Unicode
decodeURIComponent(str)	把 8 位编码 URI 组件转换为 Unicode
encodeURI(str)	把 Unicode URI 转换为 8 位编码的 URI
encodeURIComponent(str)	把 Unicode URI 组件转换为 8 位编码

类(构造函数)	
Object	提供所有对象通用的功能
Function	封装一个 ECMAScript 函数
Array	描述一个可变长的项向量
String	保存一个 Unicode 字符串
Boolean	保存一个布尔值(true 或 false)
Number	保存一个浮点数
Date	保存日期和时间
RegExp	提供正则表达式的模式匹配
Error	基本错误类型
EvalError	错误地使用 eval()时抛出
RangeError	数值溢出时抛出
ReferenceError	试图访问未定义变量时抛出
SyntaxError	eval()检测出语法错误时抛出
TypeError	参数类型错误时抛出
URIError	URI 解析失败时抛出

对象	
Math	提供数学常量和公式

图 22.1 全局对象的内置属性

① ECMAScript 标准并未说明。

可以使用 .(点)操作符调用成员函数：

```
var shape = new Shape(10, 20);
shape.translate(100, 100);
print(shape.x + ", " + shape.y + " (" + shape.manhattanPos() + ")");
```

使用这种方法，每个 Shape 实例都包含自己的 manhattanPos 和 translate 属性。由于这些属性对于所有的 Shape 实例应该是一样的，所以它们只需要保存一次，而不是保存在每个实例中。ECMAScript 允许我们使用原型法达到这个目的。原型(prototype)是一个备用的对象，提供属性的初始化集。这种方法的好处是可以随时修改原型对象，并且修改可以立即反映到使用该原型创建的所有对象中。

请看下面的例子：

```
function Shape(x, y) {
    this.x = x;
    this.y = y;
}

Shape.prototype.manhattanPos = function() {
    return Math.abs(this.x) + Math.abs(this.y);
};

Shape.prototype.translate = function(dx, dy) {
    this.x += dx;
    this.y += dy;
};
```

在 Shape 的这个版本中，我们在构造函数之外创建 manhattanPos 和 translate 属性，把它们作为 Shape.prototype 对象的属性。当实例化 Shape 时，新创建的对象保存一个指向 Shape.prototype 的内部指针。当查询一个 Shape 对象中不存在的属性值时，将会到原型中查询。因此，Shape 原型是保存可以被所有 Shape 实例共享的成员函数的理想场所。

把需要在 Shape 实例中共享的各种属性保存在原型中的方法不错，这与 C++ 的静态成员变量或者 Java 的类变量类似。这一特性对只读属性(包括成员函数)有用，因为原型是用来查询属性值的一种备用策略。然而，试图给共享变量赋新值是行不通的。取而代之的是，可以创建一个新的变量，以覆盖原型中具有相同名称的属性。这种对变量读和写的不对称访问经常使 ECMAScript 程序员新手感到困惑。

在基于类的语言中，比如 C++ 和 Java，可以使用类继承创建一系列的对象。例如，可以定义一个 Shape 类，然后从 Shape 继承出 Triangle、Square 和 Circle。在 ECMAScript 中，使用原型可以实现类似的效果。下面的例子介绍了如何定义 Circle 对象，它也是 Shape 实例：

```
function Shape(x, y) {
    this.x = x;
    this.y = y;
}

Shape.prototype.area = function() { return 0; };

function Circle(x, y, radius) {
    Shape.call(this, x, y);
    this.radius = radius;
}

Circle.prototype = new Shape;
Circle.prototype.area = function() {
    return Math.PI * this.radius * this.radius;
};
```

首先定义一个 Shape 构造函数，关联一个总是返回 0 的 area()函数。然后定义一个 Circle 构造函数，它使用 call()函数调用“基类”的构造函数，call()函数是为所有函数对象(包括构造函数)定义的，此外还添加了一个 radius 属性。在构造函数外，我们设置 Circle 的原型为一个 Shape 对象，并且用 Circle 定义的实现覆盖 Shape 的 area()函数。下面就是对应的 C++ 代码：

```
class Shape
{
public:
    Shape(double x, double y) {
        this->x = x;
        this->y = y;
    }

    virtual double area() const { return 0; }

    double x;
    double y;
};

class Circle : public Shape
{
public:
    Circle(double x, double y, double radius)
        : Shape(x, y)
    {
        this->radius = radius;
    }

    double area() const { return M_PI * radius * radius; }

    double radius;
};
```

instanceof 操作遍历原型链,以判断哪个构造函数被调用了。因而,子类实例也被认为是基类的实例:

```
var circle = new Circle(0, 0, 50);
circle instanceof Circle;          // returns true
circle instanceof Shape;           // returns true
circle instanceof Object;          // returns true
circle instanceof Array;           // returns false
```

这就是我们对 ECMAScript 的简短介绍。在接下来的章节中,将介绍如何在 C++/Qt 应用程序中使用这一语言,为终端用户提供更多的灵活性和可定制能力,或者仅仅为了提高开发进度。

22.2 使用脚本扩展 Qt 应用程序

使用 QtScript 模块,我们可以编写能够运行 ECMAScript 的 C++ 应用程序。脚本可以扩展应用程序的功能,而不需要重新编译和重新部署应用程序。我们可以把 ECMAScript 文件的硬编码集作为应用程序的一部分,也可以被新版本的应用程序所替代,或者也可以让应用程序使用任意的 ECMAScript 文件。

在 C++ 应用程序中执行脚本通常包含以下步骤:

1. 把脚本读入 QString。
2. 创建一个 QScriptEngine 对象,并设置应用程序相关的功能。
3. 执行脚本。

为了说明这一点,我们将学习如图 22.2 所示的 Calculator 应用程序。Calculator 应用程序允许用户在脚本中实现某些功能来提供一些自定义按钮。当应用程序启动时,它会访问 scripts 的子目录,寻找脚本文件,创建与这些脚本相关的计算器按钮。默认情况下,Calculator 包含以下脚本:

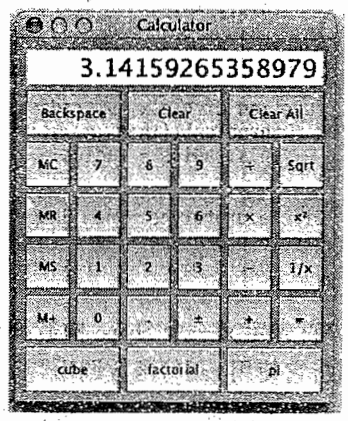


图 22.2 Calculator 应用程序

- cube.js 计算当前值的立方(x^3)。
- factorial.js 计算当前值的阶乘($x!$)。
- pi.js 用 π 的近似值覆盖当前值。

大部分 Calculator 应用程序的代码都与本书中使用的 C++ /Qt 代码属于同一类型。这里,我们仅介绍与脚本相关的代码,先从 createCustomButtons()私有函数开始,它在 Calculator 构造函数中得到调用:

```
void Calculator::createCustomButtons()
{
    QDir scriptsDir = directoryOf("scripts");
    QStringList fileNames = scriptsDir.entryList(QStringList("*.js"),
                                                 QDir::Files);
    foreach (QString fileName, fileNames) {
        QString text = fileName;
        text.chop(3);
        QToolButton *button = createButton(text,
                                           SLOT(customButtonClicked()));
        button->setStyleSheet("color: rgb(31, 63, 127)");
        button->setProperty("scriptFileName",
                            scriptsDir.absoluteFilePath(fileName));
        customButtons.append(button);
    }
}
```

createCustomButtons()函数使用 QDir 对象访问应用程序脚本子目录,寻找带 .js 扩展名的文件。它使用相同的 directoryOf()函数,第 17 章已使用过它。

对于每个 .js 文件,通过调用私有函数 createButton()函数创建 QToolButton。该函数还连接新按钮的 clicked()信号和 customButtonClicked()槽。然后,设置按钮的样式表使前景色文字为蓝色,以区别于自定义按钮和内置按钮。

调用 QObject::setProperty()为 QToolButton 动态创建一个新的 scriptFileName 属性。在 customButtonClicked()槽中使用这一属性来确定应该执行哪个脚本。

最后,把新按钮添加到 customButtons 列表中。Calculator 构造函数将列表中的定制按钮添加到窗口的网格布局中。

对于此应用程序,在它启动时访问脚本子目录一次。另一个方法是使用 QFileSystemWatcher 监听脚本的子目录,当目录内容发生改变时更新计算器,允许用户不必重新启动应用程序就可以添加新脚本和删除已存在的脚本。

```
void Calculator::customButtonClicked()
{
    QToolButton *clickedButton = qobject_cast<QToolButton *>(sender());
    QFile file(clickedButton->property("scriptFileName").toString());
    if (!file.open(QIODevice::ReadOnly)) {
        abortOperation();
        return;
    }

    QTextStream in(&file);
    in.setCodec("UTF-8");
    QString script = in.readAll();
    file.close();

    QScriptEngine interpreter;
    QScriptValue operand(&interpreter, display->text().toDouble());
    interpreter.globalObject().setProperty("x", operand);
    QScriptValue result = interpreter.evaluate(script);
    if (!result.isNumber()) {
```

```
        abortOperation();
        return;
    }

    setDisplayValue(result.toNumber());
    waitingForOperand = true;
}
```

在 customButtonClicked()槽中,首先调用 QObject::sender()确定被单击的按钮。然后,提取 scriptFileName 属性查询与该按钮相关的 .js 文件名。接下来,把文件的内容读入 script 字符串中。

ECMAScript 规范要求解释器支持 Unicode,但它没有要求使用哪个编码保存脚本。我们假定 .js 文件使用 UTF-8,这是一个纯 ASCII 的扩展集。

当脚本被读入 QString 时,我们创建一个 QScriptEngine 去执行它。一个 QScriptEngine 实例代表一个 ECMAScript 解释器,并保存当前的状态。我们可以同时拥有任意数量的 QScriptEngine,各自保存自己的状态。

在运行脚本之前,必须保证脚本可以查询到计算器当前显示的值。选择的方法是创建一个称作 x 的 ECMAScript 全局变量——或者,更准确地说,就是向解释器的全局对象中添加一个称为 x 的动态属性。在代码中,x 可以被直接使用。

设置的 x 的值的类型必须是 QScriptValue。概念上,QScriptValue 类似于 QVariant,因为它可以存储很多的数据类型,但它只能保存 ECMAScript 数据类型。

最后,使用 QScriptEngine::evaluate()运行脚本。结果是脚本返回的值,或者如果发生错误,返回一个异常对象。(在下一节中,我们将了解如何用消息框报告错误。)脚本的返回值是 return 调用的返回值;如果 return 被忽略了,那么返回值就是最后一个表达式求得的值。一旦得到了返回值,就可以检查它是否是一个数字,如果是就显示它。

对于这个例子,每次用户单击相应的按钮,脚本就会执行一次。因为这一步包含加载和解析整个脚本,通常更可取的是使用一种不同的方法,脚本不是立即执行,而是返回一个函数或者对象,它可以被再次使用。下一节将使用这一替代方法。

为了链接 QtScript 库,必须在应用程序的 .pro 文件中添加这一行:

```
QT += script
```

例子脚本很简单,就是这样一行的 pi.js:

```
return 3.14159265358979323846;
```

值得注意的是,我们忽略了计算器的 x 值。cube.js 脚本也是一行,但它使用了 x 的值:

```
return x * x * x;
```

factorial.js 脚本定义了一个函数并且调用了它:

```
function factorial(n)
{
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

return factorial(Math.floor(x));
```

标准的阶乘函数仅操作整数,因此使用了 Math.floor()函数把 x 转换为整数。

我们了解了 QtScript 模块的基础:QScriptEngine 和 QScriptValue,前者代表一个解释器和它当前的状态,后者保存一个 ECMAScript 的值。

在计算器例子中，很少涉及脚本和应用程序之间的交互：脚本从应用程序中读取一个参数，返回一个单独的值。在下面的几节中，将了解更高级的集成办法，并且介绍如何向用户报告异常。

22.3　使用脚本实现 GUI 扩展

如同前面介绍的，提供脚本计算数值很有用但有局限性。通常，我们想要在脚本中直接访问应用程序的窗口部件以及其他的组件。也可能需要组合 ECMAScript 文件和 Qt 设计师的 .ui 文件提供额外的对话框。使用这些技术，就能够主要使用 ECMAScript 来开发应用程序，很多程序员也正希望如此。

本节将介绍如图 22.3 所示的 HTML 编辑器应用程序。该应用程序是一个纯文本编辑器，使用 QSyntaxHighlighter 高亮显示 HTML 标签。该应用程序特殊的地方是它提供了一个 Scripts 菜单，它将应用程序 scripts 子目录下的以 .js 作为扩展名的脚本文件和相应的 .ui 对话框组装起来。对话框允许用户参数化他们想要执行的操作。

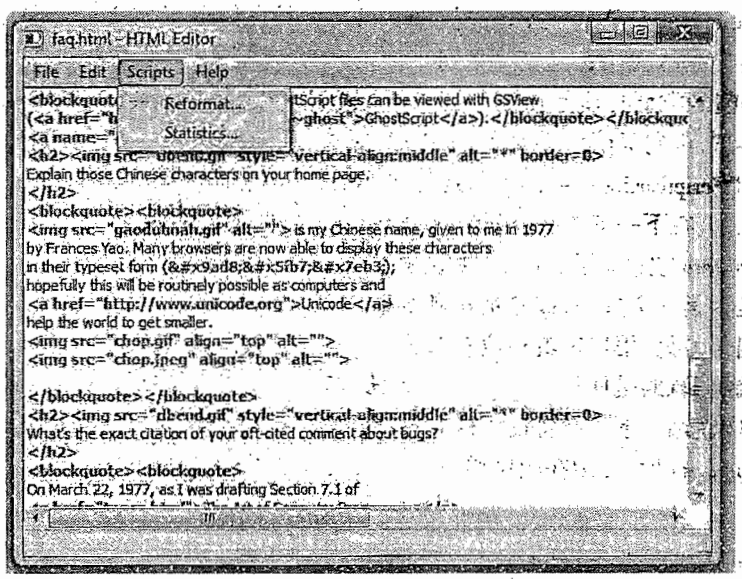


图 22.3　HTML 编辑器应用程序

我们提供了两个扩展：Statistics 对话框和 Reformat Text 对话框，如图 22.4 所示。Statistics 对话框是个纯粹的信息提供者，它统计文档中的字符、单词和行的数量，在一个模式对话框中显示总数。Reformat Text 对话框更复杂一些，它是非模态对话框，这意味着在对话框显示的情况下用户仍然可以与应用程序的主窗口进行交互。该对话框可以用来重新对齐文字、换行，以及标准化标签的语法。所有这些操作都实现在 ECMAScript 中。

该应用程序的核心是 HtmlWindow 类，它是一个 QMainWindow 的子类，使用 QTextEdit 作为中心窗口部件。这里，我们只介绍与脚本相关的代码。

当应用程序启动时，我们必须组装脚本菜单，菜单动作与 scripts 子目录下的 .js 和 .ui 文件相对应。这与前面的 Calculator 应用程序中的 createCustomButtons() 函数很相似：

```
void HtmlWindow::createScriptsMenu()
{
    scriptsMenu = menuBar()->addMenu(tr("&Scripts"));

    QDir scriptsDir = directoryOf("scripts");
```

```
    QStringList jsFileNames = scriptsDir.entryList(QStringList("*.js"),
                                                   QDir::Files);
    foreach (QString jsFileName, jsFileNames)
        createScriptAction(scriptsDir.absoluteFilePath(jsFileName));

    scriptsMenu->setEnabled(!scriptsMenu->isEmpty());
}
```

对于每个脚本,我们调用 createScriptAction()函数创建动作,并把它添加到 Scripts 菜单中。如果没有发现任何脚本,就将菜单设为无效。

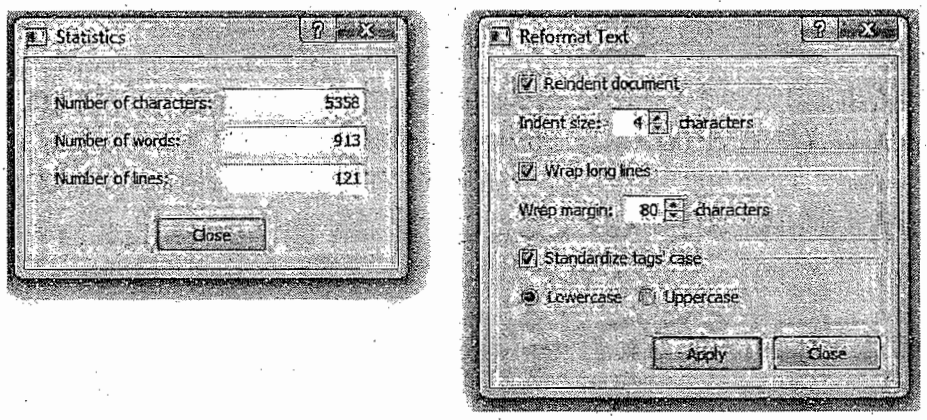


图 22.4 Statistics 和 Reformat Text 对话框

createScriptAction()函数执行以下步骤:

1. 调用并求脚本的值,保存结果对象到变量中。
2. 使用 QUiLoader 和 .ui 文件创建对话框。
3. 使脚本能够访问对话框。
4. 向脚本揭示应用程序的相关功能。
5. 创建 QAction 使用户可以访问脚本。

函数需要做很多工作,而且很长,因此将分段介绍它。

```
bool HtmlWindow::createScriptAction(const QString &jsFileName)
{
    QFile jsFile(jsFileName);
    if (!jsFile.open(QIODevice::ReadOnly)) {
        QMessageBox::warning(this, tr("HTML Editor"),
                             tr("Cannot read file %1:\n%2.")
                             .arg(strippedName(jsFileName))
                             .arg(jsFile.errorString()));
        return false;
    }

    QTextStream in(&jsFile);
    in.setCodec("UTF-8");
    QString script = in.readAll();
    jsFile.close();

    QScriptValue qsScript = interpreter.evaluate(script);
    if (interpreter.hasUncaughtException()) {
        QMessageBox messageBox(this);
        messageBox.setIcon(QMessageBox::Warning);
        messageBox.setWindowTitle(tr("HTML Editor"));
        messageBox.setText(tr("An error occurred while executing the "
                              "script %1.")
                           .arg(strippedName(jsFileName)));
```

```
        messageBox.setInformativeText(
                tr("%1.").arg(interpreter.uncaughtException()
                              .toString()));
        messageBox.setDetailedText(
                interpreter.uncaughtExceptionBacktrace().join("\n"));
        messageBox.exec();
        return false;
    }
```

首先读取 .js 文件。由于只需要一个解释器,可以使用 QScriptEngine 类型的称作 interpreter 的成员变量。我们求取脚本的值,并把它保存在名为 qsScript 的 QScriptValue 对象中。

如果脚本不能被求值(例如语法错误),QScriptEngine::hasUncaughtException()函数将返回 true。在这种情况下,可以使用 QMessageBox 报告错误。

对于本应用程序所使用的脚本,我们采用一种约定,当被求值时,每个脚本都必须返回一个 ECMAScript 对象。这个对象必须提供两个属性:一个称作 text 的字符串保存用以识别脚本的 Scripts 菜单的内容,以及一个称作 run()的函数,当用户选择 Scripts 菜单的脚本时可以被调用。我们把对象保存在 qsScript 变量中。这种方法最大的好处是只需在启动程序时读取和解析脚本一次。

```
QString uiFileName = jsFileName;
uiFileName.chop(3);
uiFileName += ".ui";

QFile uiFile(uiFileName);
if (!uiFile.open(QIODevice::ReadOnly)) {
    QMessageBox::warning(this, tr("HTML Editor"),
                         tr("Cannot read file %1:\n%2.")
                         .arg(strippedName(uiFileName))
                         .arg(uiFile.errorString()));
    return false;
}

QUiLoader loader;
QWidget *dialog = loader.load(&uiFile, this);
uiFile.close();
if (!dialog) {
    QMessageBox::warning(this, tr("HTML Editor"),
                         tr("Error loading %1.")
                         .arg(strippedName(uiFileName)));
    return false;
}
```

另外的约定是每个脚本必须有一个相对应的 .ui 文件,为脚本提供一个 GUI 对话框。.ui 文件和脚本文件的文件名前部分必须相同。

我们试图读取 .ui 文件,动态地创建一个包含所有窗口部件、布局和 .ui 文件中指定的连接的 QWidget。窗口部件的父对象作为 load()调用的第二个参数给定。如果发生错误,向用户发出警告并返回。

```
    QScriptValue qsDialog = interpreter.newQObject(dialog);
    qsScript.setProperty("dialog", qsDialog);

    QScriptValue qsTextEdit = interpreter.newQObject(textEdit);
    qsScript.setProperty("textEdit", qsTextEdit);
    QAction *action = new QAction(this);
    action->setText(qsScript.property("text").toString());
    action->setData(QVariant::fromValue(qsScript));
    connect(action, SIGNAL(triggered()),
            this, SLOT(scriptActionTriggered()));

    scriptsMenu->addAction(action);

    return true;
}
```

一旦成功读取脚本文件和用户接口文件,就可以把脚本添加到 Scripts 菜单中。但首先有几个需要注意的细节:脚本的 run()函数需要能够访问创建的对话框。此外,脚本应该能够访问 QTextEdit,它包含被编辑的 HTML 文档。

我们首先把对话框作为 QObject*添加到解释器中,解释器使用该对象来显示对话框,把它保存在 qsDialog 中。我们把 qsDialog 对象添加到 qsScript 对象中,把它作为一个新的称作 dialog 的属性。这样,脚本就可以通过新创建的 dialog 属性访问对话框,包括它的窗口部件。使用同样的技术使脚本可以访问应用程序的 QTextEdit。

最后,创建一个新的 QAction 在 GUI 中显示脚本。设置动作的文字为 qsScript 的 text 属性,动作的"data"项设置为 qsScript。最后,连接动作的 triggered()信号和定制的 scriptActionTriggered()槽,把动作添加到 Scripts 菜单中。

```
void HtmlWindow::scriptActionTriggered()
{
    QAction *action = qobject_cast<QAction *>(sender());
    QScriptValue qsScript = action->data().value<QScriptValue>();
    qsScript.property("run").call(qsScript);
}
```

当调用该槽时,需要首先查出是哪个动作得到了触发。然后,使用 QVariant::value <T>()提取出该动作的用户数据,并且把它转换为 QScriptValue,保存在 qsScript 中。接着,触发 qsScript 的 run()函数,以 qsScript 为参数,这将使 qsScript 作为 run()函数的 this 对象①。

QAction 的"data"项机制基于 QVariant。QScriptValue 类型不是 QVariant 可以识别的数据类型。幸运的是,Qt 提供了一种机制可以扩展 QVariant 处理的类型。在 htmlwindow.cpp 开始的地方,#include 的后面,有这样一行:

```
Q_DECLARE_METATYPE(QScriptValue)
```

这一行应该出现在定制的数据类型被声明之后,且只对那些有默认构造函数和复制构造函数的数据类型有效。

既然已经了解了如何加载脚本和用户接口文件,以及如何提供一个动作,用户可以触发它并运行脚本,我们可以看看脚本本身了。从 Statistics 脚本开始,因为它最容易也最短,将分段介绍它。

```
var obj = new Object;

obj.text = "&Statistics...";
```

我们创建一个新的对象,为它添加属性,把它返回给解释器。第一个属性是 text 属性,它保存 Scripts 菜单显示的文字。

```
obj.run = function() {
    var text = this.textEdit.plainText;
    this.dialog.frame.charCountLineEdit.text = text.length;
    this.dialog.frame.wordCountLineEdit.text = this.wordCount(text);
    this.dialog.frame.lineCountLineEdit.text = this.lineCount(text);
    this.dialog.exec();
};
```

创建的第二个属性是 run()函数。该函数从对话框的 QTextEdit 中读取文字,把对话框的窗口部件和计算结果结合起来,最后模态地显示对话框。

① Qt 4.4 有望提供一个 qScriptConnect()函数,允许我们创建 C++ 和脚本的连接。利用这个函数,可以直接连接 QAction 的 triggered()信号和 qsScript 的 run()函数:

```
qScriptConnect(action, SIGNAL(triggered()), qsScript, qsScript.property("run"));
```

只有当对象变量 obj 具有合适的 textEdit 和 dialog 属性时，该函数才可以工作，这就是需要在 createScriptAction()函数末尾添加它们的原因。对话框本身必须有一个窗体对象（这里是 QFrame，但类型不重要），以及三个子窗口部件——charCountLineEdit、wordCountLineEdit 和 lineCountLineEdit，它们都带有可写的 text 属性。除了 this.dialog.frame.*xxx*CountLineEdit，我们也可以写 findChild("*xxx*CountLineEdit")，它将执行递归搜索，对于修改对话框的设计更有用。

```
obj.wordCount = function(text) {
    var regExp = new RegExp("\\w+", "g");
    var count = 0;
    while (regExp.exec(text))
        ++count;
    return count;
};

obj.lineCount = function(text) {
    var count = 0;
    var pos = 0;
    while ((pos = text.indexOf("\n", pos)) != -1) {
        ++count;
        ++pos;
    }
    return count + 1;
};

return obj;
```

wordCount()和 lineCount()函数没有外部依赖，纯粹根据传入的 String 进行工作。注意，wordCount()函数使用 ECMAScript RegExp 类，而不是 Qt 的 QRegExp 类。在脚本文件的最后，return 语句确保带有 text 和 run()函数属性的对象被返回到解释器中，并可以使用。

Reformat 脚本与 Statistics 脚本依照相同的样式。接下来将介绍它。

```
var obj = new Object;

obj.initialized = false;

obj.text = "&Reformat...";

obj.run = function() {
    if (!this.initialized) {
        this.dialog.applyButton.clicked.connect(this, this.apply);
        this.dialog.closeButton.clicked.connect(this, this.dialog.close);
        this.initialized = true;
    }
    this.dialog.show();
};

obj.apply = function() {
    var text = this.textEdit.plainText;

    this.textEdit.readOnly = true;
    this.dialog.applyButton.enabled = false;

    if (this.dialog.indentGroupBox.checked) {
        var size = this.dialog.indentGroupBox.indentSizeSpinBox.value;
        text = this.reindented(text, size);
    }
    if (this.dialog.wrapGroupBox.checked) {
        var margin = this.dialog.wrapGroupBox.wrapMarginSpinBox.value;
        text = this.wrapped(text, margin);
    }
    if (this.dialog.caseGroupBox.checked) {
        var lowercase = this.dialog.caseGroupBox.lowercaseRadio.checked;
        text = this.fixedTagCase(text, lowercase);
    }
```

```
    this.textEdit.plainText = text;
    this.textEdit.readOnly = false;
    this.dialog.applyButton.enabled = true;
};

obj.reindented = function(text, size) {
    ...
};

obj.wrapped = function(text, margin) {
    ...
};

obj.fixedTagCase = function(text, lowercase) {
    ...
};

return obj;
```

我们使用相同的样式创建一个普通的对象,为它添加属性,把它返回到解释器中。除了 text 和 run()属性,还添加了 initialized 属性。第一次调用 run()时,initialized 是 false,我们设置信号-槽,连接对话框上按钮的单击事件和脚本中的函数。

这里使用了 Statistics 脚本相同的假设。我们假定有一个合适的 dialog 属性,它上面有称作 applyButton 和 closeButton 的按钮。apply()函数与对话框的窗口部件交互,特别是 Apply 按钮(使其可用和禁用),以及群组框、复选框和微调框。它还与主窗口的 QTextEdit 交互,从 QTextEdit 中获取文字信息,把格式化的结果返回到 QTextEdit 中。

我们忽略了 reindented()、wrapped()和 fixedTagCase()函数的代码,它们在脚本内部使用,因为实际的计算过程跟理解如何对 Qt 应用程序脚本化并没有关系。

现在,我们完成了如何在 C++ /Qt 应用程序中使用脚本技术的介绍,包括具有自定义对话框的应用程序。在 HTML 编辑器这样的应用程序中,脚本与应用程序对象进行交互,我们还必须考虑许可协议的问题。对于开源的应用程序,它符合开源许可协议本身的要求。对于商业应用,事情就有些复杂了。对于为商业应用编写脚本的开发者,包括应用程序的最终用户,如果他们的脚本只使用 ECMAScript 内置的类和应用程序相关的 API,或者如果他们使用 Qt API 进行较少的扩展或者修改存在的组件,这样做是免费的。但任何脚本编写者编写实现内核 GUI 功能的脚本必须拥有商业 Qt 许可协议。如果有许可协议问题,商业版 Qt 使用者应该联系 Trolltech 的销售代表。

22.4 使用脚本自动化处理任务

有时,我们使用 GUI 应用程序每次以相同的方式处理数据集。如果操作需要触发许多的菜单项,或者需要与对话框交互,不仅会变得很单调,而且还会有风险,有时会漏掉某些步骤,或者颠倒两个步骤的顺序——或许有时不会意识到已经犯了错误。一个使这一工作变得容易的方法是编写能够按动作的顺序自动执行的脚本。

在本节中,我们将展现一个 GUI 应用程序,提供一个命令行参数-script,允许用户指定一个脚本去运行。启动应用程序,执行脚本,然后结束,根本没有 GUI 的显示。

这里用来阐述这种技术的应用程序称为 Gas Pump。它可以读取加油泵的交易记录列表,并且以表格的形式来显示数据,如图 22.5 所示。

对于每项交易,加油泵都记录了日期和时间、数量、公司 ID 和卡车司机的用户 ID,以及交易状态。Gas Pump 应用程序能够以老练的方式处理数据,排序、过滤、求和,并且可以在升和加仑间进行转换。

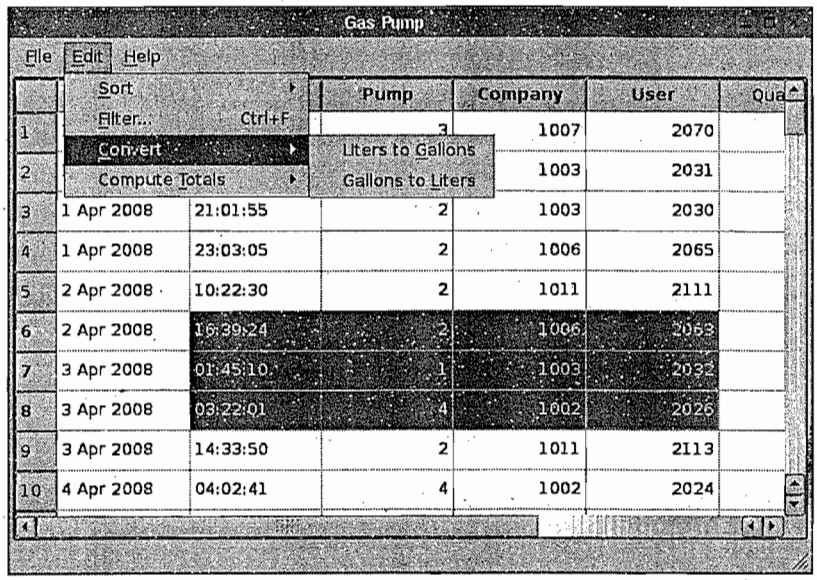


图 22.5　Gas Pump 应用程序

Gas Pump 应用程序能够以两种格式处理数据："Pump 2000"和"XML Gas Pump"，前者是以 .p20 为扩展名的普通文本格式，后者是以 .gpx 为扩展名的 XML 文本格式。应用程序可以加载和保存两种格式的数据，因此可以彼此互相转换，只需加载时使用一种格式，保存时使用另一种格式。

应用程序提供了 4 个标准的脚本：

- onlyok.js 删除所有状态不是"OK"的交易。
- p20togpx.js 把 Pump 2000 格式的文件转换为 XML Gas Pump 格式。
- tohtml.js 以 HTML 格式生成报告。
- toliters.js 从加仑到升的单位变换。

使用-script 命令行参数调用这些脚本，后面跟着脚本的名字，然后是操作的文件名。例如：

```
gaspump -script scripts/toliters.js data/2008q2.p20
```

这里，我们运行 scripts 子目录下的 toliters.js 脚本，它位于 data 子目录下 Pump 2000 格式的 2008q2.p20 数据文件。该脚本把所有的数量值从加仑转换为升，修改文件的相应位置。

Gas Pump 应用程序就像其他的 C++ /Qt 应用程序一样编写。实际上，它的代码很像第 3 章和第 4 章中的 Spreadsheet 例子。该应用程序有一个称作 PumpWindow 的 QMainWindow 的子类，它提供应用程序的框架，包括动作和菜单（菜单如图 22.6 所示）。还有一个定制的 QTableWidget，称为 PumpSpreadsheet，它用来显示数据。一个 QDialog 的子类 FilterDialog 如图 22.7 所示，用户可以使用它指定过滤条件。因为有很多的过滤条件，它们在 PumpFilter 的类中进行排序。我们简单地查看这些类，然后介绍如何向应用程序中添加脚本支持。

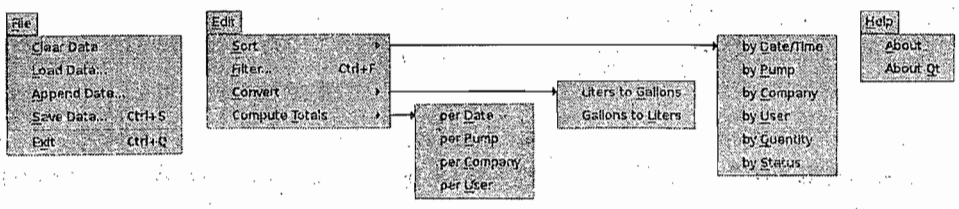


图 22.6　Gas Pump 应用程序的菜单

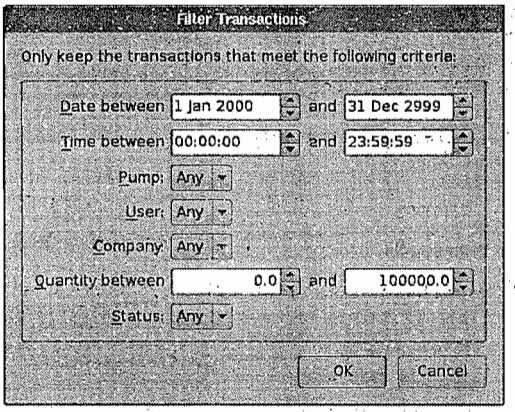


图 22.7 Filter 对话框

```
class PumpSpreadsheet : public QTableWidget
{
    Q_OBJECT
    Q_ENUMS(FileFormat Column)

public:
    enum FileFormat { Pump2000, GasPumpXml };
    enum Column { Date, Time, Pump, Company, User, Quantity, Status,
                  ColumnCount };

    PumpSpreadsheet(QWidget *parent = 0);

public slots:
    void clearData();
    bool addData(const QString &fileName, FileFormat format);
    bool saveData(const QString &fileName, FileFormat format);
    void sortByColumn(Column column,
                      Qt::SortOrder order = Qt::AscendingOrder);
    void applyFilter(const PumpFilter &filter);
    void convertUnits(double factor);
    void computeTotals(Column column);
    void setText(int row, int column, const QString &text);
    QString text(int row, int column) const;

private:
    ...
};
```

PumpSpreadsheet 保存数据,提供操作数据的函数(把函数做成了槽)。通过用户接口可以访问这些槽,编写脚本也可以使用它们。Q_ENUMS() 宏用来生成关于 FileFormat 和 Column 枚举类型的 META 标签信息,稍后再介绍它。

QMainWindow 子类 PumpWindow 有一个 loadData() 函数,它使用了一些 PumpSpreadsheet 槽:

```
void PumpWindow::loadData()
{
    QString fileName = QFileDialog::getOpenFileName(this,
                                tr("Open Data File"), ".",
                                fileFilters);
    if (!fileName.isEmpty()) {
        spreadsheet->clearData();
        spreadsheet->addData(fileName, fileFormat(fileName));
    }
}
```

PumpSpreadsheet 保存在 PumpWindow 中,作为它的成员变量,称为 spreadsheet。PumpWindow 的 filter() 槽的典型性稍逊:

```
void PumpWindow::filter()
{
    FilterDialog dialog(this);
    dialog.initFromSpreadsheet(spreadsheet);
    if (dialog.exec())
        spreadsheet->applyFilter(dialog.filter());
}
```

initFromSpreadsheet()函数把 FilterDialog 的组合框和泵、公司 ID、用户 ID 以及状态编码组合起来。当调用 exec()时，会弹出如图 22.7 所示的对话框。如果用户单击 OK，FilterDialog 的 filter()函数返回一个 PumpFilter 对象，把它传给 PumpSpreadsheet::applyFilter()。

```
class PumpFilter
{
public:
    PumpFilter();

    QDate fromDate;
    QDate toDate;
    QTime fromTime;
    QTime toTime;
    QString pump;
    QString company;
    QString user;
    double fromQuantity;
    double toQuantity;
    QString status;
};
```

PumpFilter 可以使传递过滤参数变得更容易，采用分组的方式，而不是传递 10 个单独的参数。

目前为止，我们没有看到值得惊奇的地方。不易引人注意的不同点是我们把所有想要脚本化的 PumpSpreadsheet 函数变成了公有的槽，我们也使用了 Q_ENUMS()宏。为了使 Gas Pump 脚本化，还需要做两件事。第一，必须修改 main.cpp 添加对命令行的处理，以及执行被指定的脚本。第二，必须使应用程序的功能可以被脚本使用。

QtScript 模块提供了两种常规的方式为脚本提供 C++ 的类。最简单的方式是在 QObject 类中定义一个功能，使用 QScriptEngine::newQObject()为脚本提供一个或更多的类的实例。在类(也可以是其祖先)中定义的属性和槽可以在脚本中使用。更复杂但也更灵活的方法是为类编写一个 C++ 原型以及一个构造函数，在脚本中使用 new 操作符就可以实例化这些类。Gas Pump 例子阐述了这两种方法。

在学习运行脚本的架构之前，来看 Gas Pump 提供的一个脚本。这就是完整的 onlyok.js 脚本：

```
if (args.length == 0)
    throw Error("No files specified on the command line");

for (var i = 0; i < args.length; ++i) {
    spreadsheet.clearData();
    if (!spreadsheet.addData(args[i], PumpSpreadsheet.Pump2000))
        throw Error("Error loading Pump 2000 data");

    var filter = new PumpFilter;
    filter.status = "OK";

    spreadsheet.applyFilter(filter);
    if (!spreadsheet.saveData(args[i], PumpSpreadsheet.Pump2000))
        throw Error("Error saving Pump 2000 data");

    print("Removed erroneous transactions from " + args[i]);
}
```

这个脚本依赖于两个全局变量：args 和 spreadsheet。args 变量返回命令行参数，提供-script 后面的参数。spreadsheet 变量是 PumpSpreadsheet 对象的一个引用，可以使用它做各种操作(文件格式转

换、单位变换、过滤等)。脚本调用 PumpSpreadsheet 对象的一些槽,实例化和初始化 PumpFilter 对象,使用 PumpSpreadsheet::FileFormat 枚举。

我们首先执行一个完整性检查,然后对于命令行中列出的文件,清除全局的 spreadsheet 对象,读取文件数据。假设这些文件的格式都是 Pump 2000(.p20)。对于每次成功载入文件,创建一个新的 PumpFilter 对象。我们设置过滤器的 status 属性,然后调用 PumpSpreadsheet 的 applyFilter()函数(因为它是一个槽,我们可以访问)。最后,把更新了的电子表格数据保存到原始文件中,输出一个用户信息。

其他三个脚本有相同的结构,它们包含在本书配套网站的源码中。

为了支持脚本,例如 onlyok.js,在 Gas Pump 应用程序中需要执行下面的步骤:

1. 检查-script 命令行参数。
2. 加载指定的脚本文件。
3. 在解释器中引入一个 PumpSpreadsheet 的实例。
4. 在解释器中引入命令行参数。
5. 在解释器中引入 FileFormat 和 Column 枚举。
6. 包装 PumpFilter 类,使得脚本可以访问成员变量。
7. 使 PumpFilter 对象可以在脚本中被实例化。
8. 执行脚本。

相关的代码在 main.cpp、scripting.cpp 和 scripting.h 中。首先请看 main.cpp:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringList args = QApplication::arguments();
    if (args.count() >= 3 && args[1] == "-script") {
        runScript(args[2], args.mid(3));
        return 0;
    } else if (args.count() == 1) {
        PumpWindow window;
        window.show();
        window.resize(600, 400);
        return app.exec();
    } else {
        std::cerr << "Usage: gaspump [-script myscript.js <arguments>]"
                  << std::endl;
        return 1;
    }
}
```

命令行参数通过 QApplication::arguments()函数访问,返回一个 QStringList。列表中的第一项是应用程序的名称。如果至少有三个参数,并且第二个参数是-script,就假定第三个参数是脚本名。在这种情况下,调用 runScript(),脚本名作为第一个参数,字符串列表的其他部分作为第二个参数。如果脚本已经运行,应用程序立刻终止。

如果只有一个参数——应用程序的名称,就创建并且显示 PumpWindow,以常规的方式开始应用程序的事件循环。

scripting.h 和 scripting.cpp 提供应用程序的脚本支持。这些文件定义 runScript()函数、pumpFilterConstructor()支持函数以及 PumpFilterPrototype 支持类。支持函数和支持类是 Gas Pump 应用程序专用的,但我们仍将介绍它们,因为它们揭示了使应用程序脚本化的一些要点。

下面将分段介绍 runScript()函数,因为它包含几个小细节。

```
bool runScript(const QString &fileName, const QStringList &args)
{
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly)) {
        std::cerr << "Error: Cannot read file " << qPrintable(fileName)
                  << ": " << qPrintable(file.errorString())
                  << std::endl;
        return false;
    }

    QTextStream in(&file);
    in.setCodec("UTF-8");
    QString script = in.readAll();
    file.close();
```

首先把脚本读入 QString 中。

```
    QScriptEngine interpreter;

    PumpSpreadsheet spreadsheet;
    QScriptValue qsSpreadsheet = interpreter.newQObject(&spreadsheet);
    interpreter.globalObject().setProperty("spreadsheet",
                                           qsSpreadsheet);
```

当把脚本读入 QString 以后，就创建 QScriptEngine 和 PumpSpreadsheet 实例。然后创建一个 QScriptVaule 类型的 PumpSpreadsheet 实例的引用，把它设置为解释器的全局属性，使它作为 spreadsheet 全局变量可以在脚本中被访问。所有 PumpSpreadsheet 槽和属性都可以通过 spreadsheet 变量在其他脚本中进行访问。

```
    QScriptValue qsArgs = qScriptValueFromSequence(&interpreter, args);
    interpreter.globalObject().setProperty("args", qsArgs);
```

传入 runScript()函数的 QStringList 类型的 args(可能为空)列表包含用户想要传入脚本的命令行参数。为了使脚本可以访问这些参数，必须总是创建一个 QScriptValue 来表示它们。为了把顺序容器(例如 QList <T> 或者 QVector <T>)转换为 QScriptValue，可以使用 QtScript 模块提供的 qScriptValueFromSequence()全局函数。我们把参数变量脚本化成全局的 args 变量。

```
    QScriptValue qsMetaObject =
            interpreter.newQMetaObject(spreadsheet.metaObject());
    interpreter.globalObject().setProperty("PumpSpreadsheet",
                                           qsMetaObject);
```

在 pumpspreadsheet.h 中，定义了 FileFormat 和 Column 枚举。此外，还包含 Q_ENUMS()声明指定这些枚举。在一般的 Qt 程序中很少使用 Q_ENUMS()，主要的用处是创建定制窗口部件时可以访问 Qt 设计师。但它在脚本上下文中很有用，因为可以通过注册包含它们的类的元对象在脚本中使用枚举。

通过添加 PumpSpreadsheet 的元对象作为 PumpSpreadsheet 的全局变量，FileFormat 和 Column 枚举可以在脚本中访问它们。脚本编辑器能够通过输入，比如 PumpSpreadsheet.Pump2000 查询到枚举值。

```
    PumpFilterPrototype filterProto;
    QScriptValue qsFilterProto = interpreter.newQObject(&filterProto);
    interpreter.setDefaultPrototype(qMetaTypeId<PumpFilter>(),
                                    qsFilterProto);
```

因为 ECMAScript 使用原型，不使用 C++ 的类，如果想脚本化一个定制的 C++ 类，必须采用一种迂回的方法。在 Gas Pump 例子中，我们想脚本化 PumpFilter 类。

一个方法应该是修改类本身，让它使用 Qt 的元对象系统，把它的数据成员输出成 Qt 属性。对

于 Gas Pump 例子,我们选择保持原有应用程序的完整性,创建一个包装类 PumpFilterPrototype,它可以保持和提供对 PumpFilter 的访问,我们将介绍它是怎样实现的。

setDefaultPrototype()调用告诉解释器使用 PumpFilterPrototype 实例作为 PumpFilter 对象的隐原型。该原型继承自 QObject,提供访问 PumpFilter 数据成员的 Qt 属性。

```
QScriptValue qsFilterCtor =
        interpreter.newFunction(pumpFilterConstructor,
                                qsFilterProto);
interpreter.globalObject().setProperty("PumpFilter", qsFilterCtor);
```

我们为 PumpFilter 注册一个构造函数,这样脚本编写者能够实例化 PumpFilter。在幕后,通过 PumpFilterPrototype 访问 PumpFilter 实例。

初级准备完成了,我们把脚本读入 QString,设置脚本环境,提供两个全局变量:spreadsheet 和 args。我们可以使用 PumpSpreadsheet 元对象,提供对 PumpFilter 实例的包装访问。现在可以执行脚本了。

```
    interpreter.evaluate(script);
    if (interpreter.hasUncaughtException()) {
        std::cerr << "Uncaught exception at line "
                  << interpreter.uncaughtExceptionLineNumber() << ": "
                  << qPrintable(interpreter.uncaughtException()
                                           .toString())
                  << std::endl << "Backtrace: "
                  << qPrintable(interpreter.uncaughtExceptionBacktrace()
                                           .join(", "))
                  << std::endl;
        return false;
    }

    return true;
}
```

我们照常调用 evaluate()运行脚本。如有语法错误或者其他的问题,将输出适当的错误信息。

现在查看微型支持函数 pumpFilterConstructor(),以及有点长(但很简单)的支持类 PumpFilterPrototype。

```
QScriptValue pumpFilterConstructor(QScriptContext * /* context */,
                                   QScriptEngine *interpreter)
{
    return interpreter->toScriptValue(PumpFilter());
}
```

当脚本使用 ECMAScript 语法 new PumpFilter 创建新对象时,构造函数被调用。使用 context 参数可以访问传入构造函数的参数。这里简单地忽略它们,创建一个默认的 PumpFilter 对象,包装在 QScriptValue 中。toScriptValue <T>()函数是一个模板函数,把类型 T 的参数转换为 QScriptValue。类型 T(在这里指 PumpFilter)必须使用 Q_DECLARE_METATYPE()注册。

```
Q_DECLARE_METATYPE(PumpFilter)
```

以下是原型类的定义:

```
class PumpFilterPrototype : public QObject, public QScriptable
{
    Q_OBJECT
    Q_PROPERTY(QDate fromDate READ fromDate WRITE setFromDate)
    Q_PROPERTY(QDate toDate READ toDate WRITE setToDate)
    ...
    Q_PROPERTY(QString status READ status WRITE setStatus)

public:
    PumpFilterPrototype(QObject *parent = 0);
```

```
    void setFromDate(const QDate &date);
    QDate fromDate() const;
    void setToDate(const QDate &date);
    QDate toDate() const;
    ...
    void setStatus(const QString &status);
    QString status() const;

private:
    PumpFilter *wrappedFilter() const;
};
```

原型类派生自 QObject 类和 QScriptable 类。对于每对 getter/setter 的访问，可以使用 Q_PROPERTY()。通常，使用 Q_PROPERTY() 仅仅为了访问集成到 Qt 设计师的定制窗口部件的属性，但它们在脚本环境中也很有用。如果函数被脚本使用，就可以把它们变成公有的槽或者属性。

所有的访问器都类似，因此只介绍一对典型的例子：

```
void PumpFilterPrototype::setFromDate(const QDate &date)
{
    wrappedFilter()->fromDate = date;
}

QDate PumpFilterPrototype::fromDate() const
{
    return wrappedFilter()->fromDate;
}
```

以下是 wrappedFilter()私有函数：

```
PumpFilter *PumpFilterPrototype::wrappedFilter() const
{
    return qscriptvalue_cast<PumpFilter *>(thisObject());
}
```

QScriptable::thisObject()函数返回与解释器当前执行的函数相连的 this 对象。它返回一个 QScriptValue，我们把它强制转换为它代表的 C++/Qt 类型 PumpFilter *。只有当使用 Q_DECLARE_METATYPE() 注册 PumpFilter * 时，这种强制转换才有效：

```
Q_DECLARE_METATYPE(PumpFilter *)
```

最后是 PumpFilterPrototype 构造函数：

```
PumpFilterPrototype::PumpFilterPrototype(QObject *parent)
    : QObject(parent)
{
}
```

在本例中，我们不让脚本编写者实例化它们自己的 PumpSpreadsheet 对象，而是提供一个全局的单例对象 spreadsheet 供它们使用。为了允许脚本编写者能够自己实例化 PumpSpreadsheet，需要注册一个 pumpSpreadsheetConstructor()函数，正如 PumpFilter 所实现的。

在 Gas Pump 例子中，足可以提供脚本，访问应用程序窗口部件(例如对 PumpSpreadsheet 的访问)以及用户定制数据类(例如 PumpFilter)。尽管在 Gas Pump 例子中不需要，有时使脚本访问 C++ 的函数也很有用。例如，下面是一个 C++ 定义的简单函数，脚本也可以访问它：

```
QScriptValue square(QScriptContext *context, QScriptEngine *interpreter)
{
    double x = context->argument(0).toNumber();
    return QScriptValue(interpreter, x * x);
}
```

打算给脚本使用的函数需要这样定义：

```
QScriptValue myFunc(QScriptContext *context, QScriptEngine *interpreter)
```

函数的参数可以使用 QScriptContext::argument()函数访问。返回值是 QScriptValue,我们使用 QScriptEngine 作为第一个参数创建它。

接下来的例子更详细一些:

```
QScriptValue sum(QScriptContext *context, QScriptEngine *interpreter)
{
    QScriptValue unaryFunc;
    int i = 0;

    if (context->argument(0).isFunction()) {
        unaryFunc = context->argument(0);
        i = 1;
    }

    double result = 0.0;
    while (i < context->argumentCount()) {
        QScriptValue qsArg = context->argument(i);
        if (unaryFunc.isValid()) {
            QScriptValueList qsArgList;
            qsArgList << qsArg;
            qsArg = unaryFunc.call(QScriptValue(), qsArgList);
        }
        result += qsArg.toNumber();
        ++i;
    }
    return QScriptValue(interpreter, result);
}
```

sum()函数可以用两种方法调用。一种简单的方法是用数字作为参数调用它。在这种情况下,unaryFunc 是一个无效的 QScriptValue,执行的操作是简单地把给定的数字相加并返回结果。复杂一点的方式是用 ECMAScript 函数作为第一个参数调用它,然后是任意的数字参数。在这种情况下,给定的函数针对每个数字调用一次,这些调用的结果相加并返回。我们在本章的第一节见过使用 ECMAScript 编写的相同的函数。使用 C++ 而不是 ECMAScript 实现底层的功能有时可以带来很大的性能提升。

在脚本调用 C++ 函数之前,必须使用 newFunction()和 setProperty(),使函数可以被解释器使用:

```
QScriptValue qsSquare = interpreter.newFunction(square);
interpreter.globalObject().setProperty("square", qsSquare);

QScriptValue qsSum = interpreter.newFunction(sum);
interpreter.globalObject().setProperty("sum", qsSum);
```

我们已经把 square()和 sum()设置为解释器的全局函数。现在,就可以在脚本中使用它们,如下所示:

```
interpreter.evaluate("print(sum(1, 2, 3, 4, 5, 6));");
interpreter.evaluate("print(sum(square, 1, 2, 3, 4, 5, 6));");
```

这就是使用 QtScript 模块脚本化 Qt 应用程序的介绍。该模块提供了大量的文档,包括总体介绍,以及相关类的详细描述,包括 QScriptContext、QScriptEngine、QScriptValue 和 QScriptable,都值得读一读。

第23章　平台相关特性

在这一章，将会看到一些 Qt 程序员可用的平台相关选项。我们先从如何访问一些本地应用程序编程接口(API)开始，例如 Windows 上的 Win32 应用程序编程接口、Mac OS X 上的 Carbon，以及 X11 上的 Xlib。然后，会转到对 ActiveQt 扩展的奥妙探寻之上，以说明如何在 Qt/Windows 应用程序中使用 ActiveX 控件，以及如何创建一些被当成是 ActiveX 服务器的应用程序。在最后一节，将会阐述如何让 Qt 应用程序与 X11 下的会话管理器一起协同工作的原理。

除了在这里介绍的这些特性之外，Trolltech 还提供了多个与平台相关的 Qt Solutions，其中包括用于简化 Motif/Xt 和 MFC 应用程序向 Qt 移植的 Qt/Motif 和 Qt/MFC 移植框架(migration framework)。对于 Tcl/Tk 应用程序，提供了一个类似的 froglogic 扩展，并且可以从 Klarälvdalens Datakonsult 获取一个 Microsoft Windows 资源转换器。更多详细的信息可以参考以下网页：

- http://www.trolltech.com/products/qt/addon/solutions/catalog/4/
- http://www.froglogic.com/tq/
- http://www.kdad.net/knut/

对于嵌入式开发，Trolltech 提供了 Qtopia 应用程序平台，会在第 24 章说明它。

一些预计与平台相关的 Qt 功能已经可以采用与平台无关的方式实现了。例如，在 Windows、UNIX 和 Mac OS X 平台，都有一个 Qt Solution 可以用来创建不同的服务(新进程)。

23.1　连接本地的应用程序编程接口

Qt 完善的 API 可以满足所有平台上的绝大多数需求，但是在某些情况下，我们也许想使用底层的、与平台相关的 API。在这一节中，将说明如何在 Qt 所支持的不同平台上使用平台本地的 API 来完成那些独特的任务。

对于每一个平台，QWidget 都提供了一个可以返回窗口 ID 或者其句柄的 winId()函数。QWidget 还提供了一个称为 find()的静态函数，它可以根据一个特定的窗口 ID 返回该窗口的 QWidget。我们可以把这个 ID 传递给本地的 API 来获得与平台相关的效果。例如，下列代码使用 winId()和 Mac OS X(Carbon)上的函数把一个工具窗口的标题栏移动到了该窗口的左侧(如图 23.1 所示)：



图 23.1　标题栏在侧边的 Mac OS X工具窗口

```
#ifdef Q_WS_MAC
    ChangeWindowAttributes(HIViewGetWindow(HIViewRef(toolWin.winId())),
                           kWindowSideTitlebarAttribute,
                           kWindowNoAttributes);
#endif
```

在 X11 上，以下给出了应当如何修改一个窗口属性的代码片段：

```
#ifdef Q_WS_X11
    Atom atom = XInternAtom(QX11Info::display(), "MY_PROPERTY", False);
    long data = 1;
```

```
    XChangeProperty(QX11Info::display(), window->winId(), atom, atom,
                    32, PropModeReplace,
                    reinterpret_cast<uchar *>(&data), 1);
#endif
```

这里把与平台相关的代码包围起来的 # ifdef 和 # endif 标记,将可以确保这个应用程序仍旧能够在其他平台上编译通过。

对于一个只用于 Windows 的应用程序来说,我们在这里给出了一个例子,说明了如何使用 GDI 调用在一个 Qt 窗口部件上进行绘制:

```
void GdiControl::paintEvent(QPaintEvent * /* event */)
{
    RECT rect;
    GetClientRect(winId(), &rect);
    HDC hdc = GetDC(winId());

    FillRect(hdc, &rect, HBRUSH(COLOR_WINDOW + 1));
    SetTextAlign(hdc, TA_CENTER | TA_BASELINE);
    TextOutW(hdc, width() / 2, height() / 2, text.utf16(), text.size());

    ReleaseDC(winId(), hdc);
}
```

要使这段代码能够工作,还必须重新实现 QPaintDevice::paintEngine(),以便可以返回一个 null 指针,并且还需要在这个窗口部件的构造函数中设置 Qt::WA_PaintOnScreen 属性。

下面的例子,说明了在一个使用 QPaintEngine 的 getDC()和 releaseDC()函数的绘制事件处理器中,如何组合使用 QPainter 和 GDI 调用:

```
void MyWidget::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    painter.fillRect(rect().adjusted(20, 20, -20, -20), Qt::red);
#ifdef Q_WS_WIN
    HDC hdc = painter.paintEngine()->getDC();
    Rectangle(hdc, 40, 40, width() - 40, height() - 40);
    painter.paintEngine()->releaseDC();
#endif
}
```

像这样混合使用 QPainter 和 GDI 调用,有时可能会导致一些不可预料的后果,在 QPainter 调用出现在 GDI 调用之后的时候尤其如此,因为 QPainter 会对底层的绘制层(drawing layer)的状态做出一些假设。

Qt 会定义下列 4 个窗口系统符号中的一个:Q_WS_WIN、Q_WS_X11、Q_WS_MAC 和 Q_WS_QWS(Qtopia)。在各个应用程序中使用它们之前,必须至少包含了一个 Qt 头文件。Qt 还提供了许多预处理器符号,以用来识别操作系统:

- Q_OS_AIX
- Q_OS_BSD4
- Q_OS_BSDI
- Q_OS_CYGWIN
- Q_OS_DGUX
- Q_OS_DYNIX
- Q_OS_FREEBSD
- Q_OS_HPUX
- Q_OS_HURD
- Q_OS_IRIX
- Q_OS_LINUX
- Q_OS_LYNX
- Q_OS_MAC
- Q_OS_NETBSD
- Q_OS_OPENBSD
- Q_OS_QS2EMS
- Q_OS_QSF
- Q_OS_QNX6
- Q_OS_QNX
- Q_OS_RELIANT
- Q_OS_SCO
- Q_OS_SOLARIS
- Q_OS_ULTRIX
- Q_OS_UNIXWARE
- Q_OS_WIN32
- Q_OS_WIN64

我们可以假定,这些符号最多只定义了它们中的一个。为了方便,当检测到定义了 Win32 或者 Win64 的时候,Qt 还会定义 Q_OS_WIN,并且在检测到任何一种基于 UNIX 的操作系统(包括

Linux 和 Mac OS X)时,Qt 就会定义 Q_OS_UNIX。在程序运行时,我们可以通过 QSysInfo::WindowsVersion 或者 QSysInfo::MacintoshVersion 来区分 Windows(2000、ME 等)或者 Mac OS X(10.2、10.3 等)的不同版本。

除了这些操作系统和窗口系统宏之外,Qt 还有一系列用于编译器的宏。例如,如果编译器是 Microsoft Visual C++,那么就会定义宏 Q_CC_MSVC。在避开编译器中的 bug 时,这些宏会显得非常有用。

几个与 Qt 的图形用户界面相关的类提供了一些与平台相关的函数,这些函数可以返回这个底层对象的一些底层句柄。图 23.2 列出了这些函数。

Mac OS X	
ATSFontFormatRef	QFont::handle()
CGImageRef	QPixmap::macCGHandle()
GWorldPtr	QPixmap::macQDAlphaHandle()
GWorldPtr	QPixmap::macQDHandle()
RgnHandle	QRegion::handle()
HIViewRef	QWidget::winId()

Windows	
HCURSOR	QCursor::handle()
HDC	QPaintEngine::getDC()
HDC	QPrintEngine::getPrinterDC()
HFONT	QFont::handle()
HPALETTE	QColormap::hPal()
HRGN	QRegion::handle()
HWND	QWidget::winId()

X11	
Cursor	QCursor::handle()
Font	QFont::handle()
Picture	QPixmap::x11PictureHandle()
Picture	QWidget::x11PictureHandle()
Pixmap	QPixmap::handle()
QX11Info	QPixmap::x11Info()
QX11Info	QWidget::x11Info()
Region	QRegion::handle()
Screen	QCursor::x11Screen()
SmcConn	QSessionManager::handle()
Window	QWidget::handle()
Window	QWidget::winId()

图 23.2　一些用于访问底层句柄的与平台相关的函数

在 X11 上,QPixmap::x11Info()和 QWidget::x11Info()返回一个 QX11Info 对象,这个对象提供了各式各样的指针或者句柄,比如 display()、screen()、colormap()和 visual()等。例如,我们可以使用这些函数在一个 QPixmap 或者 QWidget 上创建一个 X11 的图像上下文。

经常需要与其他工具包或者库进行交互的许多 Qt 应用程序必须在将一些底层事件(在 X11 上是 XEvent,在 Windows 上是 MSG,在 Mac OS X 上是 EventRef,在 Qtopia 上是 QWSEvent)转换成 QEvent 之前能够先访问这些底层事件。我们可以通过子类化 QApplication 并且重新实现相应的与平台相关的事件过滤器来完成这一工作,即 x11EventFilter()、winEventFilter()、macEventFilter()和 qwsEventFilter()四者之一。作为替代方法,还可以通过重新实现 x11Event()、winEvent()、macEvent()和 qwsEvent()中的一个函数来访问这些与平台相关的事件,而这些事件会被发送到某个给定的 QWidget 中。对于处理 Qt 通常忽略的特定事件类型,例如像游戏手柄事件,这种做法会很有用。

有关与平台相关的更多信息,包括在不同平台上如何配置 Qt 应用程序等问题,可以参考网页 http://doc.trolltech.com/4.3/winsystem.html。

23.2 在 Windows 上使用 ActiveX

Microsoft 的 ActiveX 技术允许应用程序与其他应用程序或者库提供的用户接口组件一起工作。它构建于 Microsoft 的 COM 基础上,为使用组件的应用程序定义了一套接口,并且为提供组件的应用程序和库提供了另外一套接口。

Qt/Windows 桌面版(Qt/Windows Desktop Edition)提供了 ActiveQt 框架,用以为 ActiveX 和 Qt 提供完美结合。ActiveQt 由两个模块组成:

- QAxContainer 模块允许我们使用 COM 对象并且可以在 Qt 应用程序中嵌入 ActiveX 控件。
- QAxServer 模块允许我们导出使用 Qt 编写的自定义的 COM 对象和 ActiveX 控件。

第一个例子将会在一个使用了 QAxContainer 模块的 Qt 应用程序中嵌入一个 Windows Media Player(如图 23.3所示)。这个 Qt 应用程序在 Windows Media Player 的 ActiveX 控件上添加了一个 Open 按钮、一个 Play/Pause 按钮、一个 Stop 按钮以及一个滑动条。



图 23.3 Media Player 应用程序

这个应用程序主窗口的类型是 PlayerWindow:

```
class PlayerWindow : public QWidget
{
    Q_OBJECT
    Q_ENUMS(ReadyStateConstants)

public:
```

```
    enum PlayStateConstants { Stopped = 0, Paused = 1, Playing = 2 };
    enum ReadyStateConstants { Uninitialized = 0, Loading = 1,
                               Interactive = 3, Complete = 4 };

    PlayerWindow();

protected:
    void timerEvent(QTimerEvent *event);

private slots:
    void onPlayStateChange(int oldState, int newState);
    void onReadyStateChange(ReadyStateConstants readyState);
    void onPositionChange(double oldPos, double newPos);
    void sliderValueChanged(int newValue);
    void openFile();

private:
    QAxWidget *wmp;
    QToolButton *openButton;
    QToolButton *playPauseButton;
    QToolButton *stopButton;
    QSlider *seekSlider;
    QString fileFilters;
    int updateTimer;
};
```

PlayerWindow 类继承了 QWidget。这里的 Q_ENUMS() 宏(就像下面的 Q_OBJECT 一样)是必需的,用来告诉 moc:在 onReadyStateChange()槽中使用的 ReadyStateConstants 类型是一个枚举类型。在 private 段,我们声明了一个 QAxWidget * 的成员变量。

```
PlayerWindow::PlayerWindow()
{
    wmp = new QAxWidget;
    wmp->setControl("{22D6F312-B0F6-11D0-94AB-0080C74C7E95}");
```

在构造函数中,我们从创建一个 QAxWidget 对象封装 Windows Media Player 的 ActiveX 控件开始。QAxContainer 模块由三个类组成:QAxObject 封装一个 COM 对象,QAxWidget 封装一个 ActiveX 控件,而 QAxBase 则为 QAxObject 和 QAxWidget 实现了 COM 的核心功能。这三个类之间的关系如图 23.4 所示。

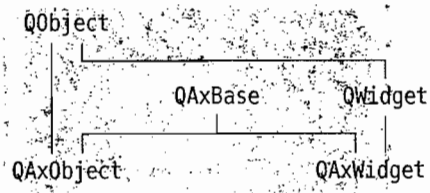


图 23.4　QAxContainer 模块的继承树

我们使用 Windows Media Player 6.4 控件的类的 ID 作为参数,对 QAxWidget 调用 setControl()。这样将会创建一个所需组件的实例。从那时起,这个 ActiveX 控件的所有属性、事件和方法都可以通过 QAxWidget 对象而作为 Qt 的属性、信号和槽来加以使用。

如图 23.5 中总结的那样,COM 数据类型可以自动转换为相应的 Qt 类型。例如,一个类型为 VARIANT_BOOL 的输入参数可以转换成一个 bool 变量,并且一个类型为 VARIANT_BOOL 的输出参数也可以转换成一个 bool & 变量。如果结果类型是一个 Qt 类(比如像 QString、QDateTime 等),那么输入参数的类型将会是一个常量引用(例如,const QString &)。

要得到带 Qt 数据类型的 QAxObject 或者 QAxWidget 中可用的属性、信号和槽列表,可以调用 QAxBase::generateDocumentation(),或者在命令行中使用 Qt 的 dumpdoc 工具,该工具位于 Qt 的 tools\activeqt\dumpdoc 目录中。

COM 类型	Qt 类型
VARIANT_BOOL	bool
char, short, int, long	int
unsigned char, unsigned short, unsigned int, unsigned long	uint
float, double	double
CY	qlonglong, qulonglong
BSTR	QString
DATE	QDateTime, QDate, QTime
OLE_COLOR	QColor
SAFEARRAY(VARIANT)	QList<QVariant>
SAFEARRAY(BSTR)	QStringList
SAFEARRAY(BYTE)	QByteArray
VARIANT	QVariant
IFontDisp *	QFont
IPictureDisp *	QPixmap
用户定义类型	QRect, QSize, QPoint

图 23.5 COM 类型和 Qt 类型之间的关系

我们继续查看 PlayerWindow 的构造函数:

```
wmp->setProperty("ShowControls", false);
wmp->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
connect(wmp, SIGNAL(PlayStateChange(int, int)),
        this, SLOT(onPlayStateChange(int, int)));
connect(wmp, SIGNAL(ReadyStateChange(ReadyStateConstants)),
        this, SLOT(onReadyStateChange(ReadyStateConstants)));
connect(wmp, SIGNAL(PositionChange(double, double)),
        this, SLOT(onPositionChange(double, double)));
```

在调用 QAxWidget::setControl()之后,通过调用 QObject::setProperty(),可以把这个 Windows Media Player 的 ShowControls 属性设置为 false,因为我们自己提供了用来操作这个组件的按钮。函数 QObject::setProperty()既可以用于 COM 属性又可以用于普通的 Qt 属性。它的第二个参数的类型是 QVariant。

接下来调用 setSizePolicy(),让这个 ActiveX 控件占有布局中所有可用的空间,并且把 COM 组件中的三个 ActiveX 事件连接到三个槽中。

```
    ...
    stopButton = new QToolButton;
    stopButton->setText(tr("&Stop"));
    stopButton->setEnabled(false);
    connect(stopButton, SIGNAL(clicked()), wmp, SLOT(Stop()));
    ...
}
```

除了把一些 Qt 信号和这个 COM 对象提供的槽[Play()、Pause()和 Stop()]连接起来以外,PlayerWindow 构造函数中的其余部分都与我们在平常模式下的情况一样。由于这些按钮的功能实现都有相似性,所以这里只给出了 Stop 按钮的实现代码。

让我们离开这个构造函数,来看一看 timerEvent()函数:

```
void PlayerWindow::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == updateTimer) {
        double curPos = wmp->property("CurrentPosition").toDouble();
        onPositionChange(-1, curPos);
    } else {
```

```
        QWidget::timerEvent(event);
    }
}
```

当正在播放一个多媒体片断的时候，每隔一定时间就会调用 timerEvent()函数一次。我们使用它推进滑动条的滑块。通过调用 ActiveX 控件上的 property()获得 CurrentPosition 属性的 QVariant 类型的值，然后调用 toDouble()把它转换成 double 值，就可以实现滑块的推进。然后，我们调用 onPositionChange()来执行更新。

我们将不会再去查看其余的代码了，因为它们中的绝大多数都不直接和 ActiveX 相关，并且里面也没有任何以前没有涉及过的东西。在本书的例子中包含了所有这些代码。

在 .pro 文件中，需要使用下面这一项来连接这个 QAxContainer 模块：

```
CONFIG += qaxcontainer
```

在处理多个 COM 对象时，经常需要能够直接调用一个 COM 方法(而不是把它连接到一个 Qt 信号上)。要做到这一点，最容易的方法就是使用这个方法的名字和签名作为调用 QAxBase::dynamicCall()的第一个形式参数，并且把这个方法的实际参数作为额外参数。例如：

```
wmp->dynamicCall("TitlePlay(uint)", 6);
```

这个 dynamicCall()函数最多可以带 8 个 QVariant 类型的参数，并且它可以返回一个 QVariant。如果需要使用这种方法传递 IDispatch * 或者 IUnkown * ，就可以把这个组件封装到一个 QAxObject 中，并且对它调用 asVariant()，以将其转换成一个 QVariant。如果需要调用能够返回 IDispatch * 或者 IUnkown * 的 COM 方法，或者如果需要访问一个具有上述类型之一的 COM 属性，那么就可以使用 querySubObject()来替代：

```
QAxObject *session = outlook.querySubObject("Session");
QAxObject *defaultContacts =
        session->querySubObject("GetDefaultFolder(OlDefaultFolders)",
                                "olFolderContacts");
```

如果我们希望调用一些函数，而这些函数的参数列表中还有一些不支持的数据类型，那么就可以使用 QAxBase::queryInterface()来取得 COM 的接口并且直接调用这个函数。就像往常使用 COM 一样，在我们已经完成了对 COM 接口的使用时，必须调用 Release()。如果需要经常调用这样的函数，那么可以子类化 QAxObject 或者 QAxWidget，并且再提供一些封装这些 COM 接口调用的成员函数即可。需要注意的是，这些 QAxObject 和 QAxWidget 的子类不能定义它们自己的属性、信号或者槽。

现在查看一下 QAxServer 模块。这个模块使我们能够把标准的 Qt 程序变成 ActiveX 服务器。这个服务器既可以是一个共享库，也可以是一个独立运行的应用程序。通常把构建为共享库的服务器称为进程内服务器(in-process server)，而把可以独立运行的应用程序称为进程外服务器(out-of-process server)。

第一个 QAxServer 例子就是一个进程内服务器，它提供一个可以显示左右弹跳的球的窗口部件。我们还将看到如何把这样的窗口部件嵌入到 Internet Explorer 浏览器中。

这里给出的是 AxBouncer 窗口部件的类定义的开头部分：

```
class AxBouncer : public QWidget, public QAxBindable
{
    Q_OBJECT
    Q_ENUMS(SpeedValue)
    Q_PROPERTY(QColor color READ color WRITE setColor)
    Q_PROPERTY(SpeedValue speed READ speed WRITE setSpeed)
    Q_PROPERTY(int radius READ radius WRITE setRadius)
    Q_PROPERTY(bool running READ isRunning)
```

如图 23.6 所示的 AxBouncer 同时继承了 QWidget 和 QAxBindable。类 QAxBindable 在这个窗口部件和 ActiveX 客户端之间提供了一个接口。任何 QWidget 都可以当作一个 ActiveX 控件导出,但是通过子类化 QAxBindable,当一个属性的值发生变化,就可以通知该客户端,并且可以实现 COM 接口来补充那些已经由 QAxServer 实现过的接口。

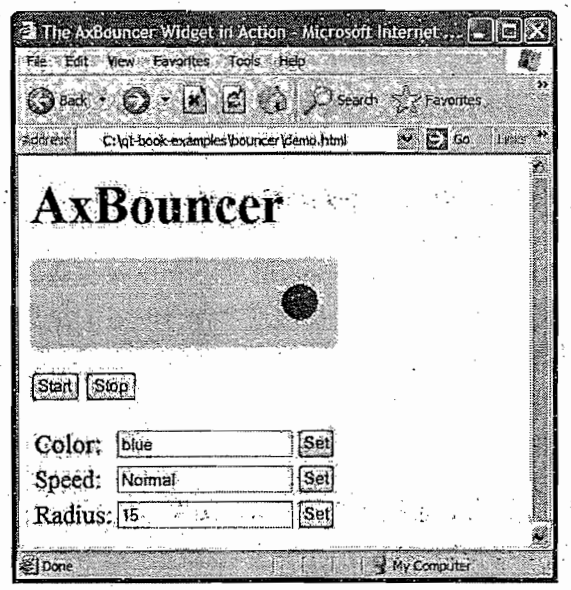


图 23.6 Internet Explorer 中的 AxBouncer 窗口部件

当包括一个源于 QObject 类的多重继承的时候,就必须总是把这个起源于 QObject 的类放在第一位,以便 moc 可以快速识别它。

我们声明了三个读写属性和一个只读属性。Q_ENUMS() 宏是必需的,它用来告诉 moc 这个 SpeedValue 类型是一个枚举类型。在这个类的 public 段中,声明了这个枚举类型:

```
public:
    enum SpeedValue { Slow, Normal, Fast };

    AxBouncer(QWidget *parent = 0);

    void setSpeed(SpeedValue newSpeed);
    SpeedValue speed() const { return ballSpeed; }
    void setRadius(int newRadius);
    int radius() const { return ballRadius; }
    void setColor(const QColor &newColor);
    QColor color() const { return ballColor; }
    bool isRunning() const { return myTimerId != 0; }

    QSize sizeHint() const;
    QAxAggregated *createAggregate();

public slots:
    void start();
    void stop();

signals:
    void bouncing();
```

AxBouncer 构造函数是一个可用于窗口部件的标准构造函数,它有一个 parent 参数。将用于导出组件的 QAXFACTORY_DEFAULT() 宏希望带一个有这样签名的构造函数。

createAggregate() 函数是从 QAxBindable 中重新实现的。我们将在稍后解释它。

```
protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);

private:
    int intervalInMilliseconds() const;

    QColor ballColor;
    SpeedValue ballSpeed;
    int ballRadius;
    int myTimerId;
    int x;
    int delta;
};
```

在这个类的 protected 和 private 段中，应当会和标准 Qt 窗口部件中的内容一致。

```
AxBouncer::AxBouncer(QWidget *parent)
    : QWidget(parent)
{
    ballColor = Qt::blue;
    ballSpeed = Normal;
    ballRadius = 15;
    myTimerId = 0;
    x = 20;
    delta = 2;
}
```

这个 AxBouncer 构造函数初始化了这个类的一些私有变量。

```
void AxBouncer::setColor(const QColor &newColor)
{
    if (newColor != ballColor && requestPropertyChange("color")) {
        ballColor = newColor;
        update();
        propertyChanged("color");
    }
}
```

这个 setColor()函数设置了 color 属性的值。它调用 update()来重新绘制这个窗口部件。

与众不同的部分是 requestPropertyChange()调用和 propertyChanged()调用。这两个函数是从 QAxBindable 中继承而来的，并且无论何时只要我们一改变某个属性，它们就应当立刻得到调用。requestPropertyChange()函数会要求客户端允许改变一个属性值，并且如果客户端允许改变的话，它就会返回 true。而 propertyChanged()函数会通知客户端：这个属性已经改变了。

setSpeed()和 setRadius()这两个设置器也都遵循这一模式，并且在 start()和 stop()槽中也是这样的，因为它们会改变这个 running 属性的值。

还剩下一个有趣的 AxBouncer 成员函数：

```
QAxAggregated *AxBouncer::createAggregate()
{
    return new ObjectSafetyImpl;
}
```

createAggregate()函数是从 QAxBindable 中重新实现的。它允许我们实现 QAxServer 模块中还没有实现的 COM 接口或者绕开 QAxServer 默认的那些 COM 接口。这里，我们使用它提供 IObjectSafety 接口，Internet Explorer 会使用它访问一个组件的安全选项。这就是去除 Internet Explorer 中声明狼藉的“Object not safe for scripting”(这个对象对于脚本是不安全的)错误消息的一个标准窍门。

以下是这个类的定义，其中实现了 IObjectSafety 接口：

```
class ObjectSafetyImpl : public QAxAggregated, public IObjectSafety
{
public:
```

```
    long queryInterface(const QUuid &iid, void **iface);

    QAXAGG_IUNKNOWN

    HRESULT WINAPI GetInterfaceSafetyOptions(REFIID riid,
            DWORD *pdwSupportedOptions, DWORD *pdwEnabledOptions);
    HRESULT WINAPI SetInterfaceSafetyOptions(REFIID riid,
            DWORD pdwSupportedOptions, DWORD pdwEnabledOptions);
};
```

ObjectSafetyImpl 类既继承了 QAxAggregated 又继承了 IObjectSafety。QAxAggregated 类是实现其他附加 COM 接口的一个抽象基类。由 QAxAggregated 扩展的 COM 对象是可以通过 controllingUnknown()访问的。COM 对象是在后台中由这个 QAxServer 模块创建的。

QAXAGG_IUNKNOWN宏提供了 QueryInterface()、AddRef()和 Release()的标准实现。这些实现只是对正在控制的 COM 对象简单地调用一些相同的函数。

```
long ObjectSafetyImpl::queryInterface(const QUuid &iid, void **iface)
{
    *iface = 0;
    if (iid == IID_IObjectSafety) {
        *iface = static_cast<IObjectSafety *>(this);
    } else {
        return E_NOINTERFACE;
    }
    AddRef();
    return S_OK;
}
```

queryInterface()函数在 QAxAggregated 中只是一个纯虚函数。正在控制的 COM 对象可以调用它允许对 QAxAggregated 子类提供的接口进行访问。我们必须为没有实现的接口和 IUnknown 返回 E_NOINTERFACE。

```
HRESULT WINAPI ObjectSafetyImpl::GetInterfaceSafetyOptions(
        REFIID /* riid */, DWORD *pdwSupportedOptions,
        DWORD *pdwEnabledOptions)
{
    *pdwSupportedOptions = INTERFACESAFE_FOR_UNTRUSTED_DATA
                         | INTERFACESAFE_FOR_UNTRUSTED_CALLER;
    *pdwEnabledOptions = *pdwSupportedOptions;
    return S_OK;
}

HRESULT WINAPI ObjectSafetyImpl::SetInterfaceSafetyOptions(
        REFIID /* riid */, DWORD /* pdwSupportedOptions */,
        DWORD /* pdwEnabledOptions */)
{
    return S_OK;
}
```

GetInterfaceSafetyOptions()和 SetInterfaceSafetyOptions()函数都是在 IObjectSafety 中声明的。我们实现它们是要告诉世人,我们的对象对于脚本是安全的。

现在看一下 main.cpp 文件:

```
#include <QAxFactory>

#include "axbouncer.h"

QAXFACTORY_DEFAULT(AxBouncer,
                   "{5e2461aa-a3e8-4f7a-8b04-307459a4c08c}",
                   "{533af11f-4899-43de-8b7f-2ddf588d1015}",
                   "{772c14a5-a840-4023-b79d-19549ece0cd9}",
                   "{dbce1e56-70dd-4f74-85e0-95c65d86254d}",
                   "{3f3db5e0-78ff-4e35-8a5d-3d3b96c83e09}")
```

QAXFACTORY_DEFAULT()宏可以导出 ActiveX 控件。我们可以把它用作仅仅导出一个控件的 ActiveX 服务器。在本节的下一个例子中,将会给出如何使用它来导出许多 ActiveX 控件的示例。

QAXFACTORY_DEFAULT()宏的第一个参数是要导出的 Qt 类的名字。它也是下面将要导出的控件的名字。其他 5 个参数分别是类的 ID、接口 ID、事件接口 ID、类型库 ID 以及应用程序 ID。我们可以使用一些像 guidgen 或者 uuidgen 这样的标准工具来生成这些标识符。因为服务器是一个库,所以不需要 main()函数。

以下是这个进程内的 ActiveX 服务器的 .pro 文件:

```
TEMPLATE      = lib
CONFIG       += dll qaxserver
HEADERS       = axbouncer.h \
                objectsafetyimpl.h
SOURCES       = axbouncer.cpp \
                main.cpp \
                objectsafetyimpl.cpp
RC_FILE       = qaxserver.rc
DEF_FILE      = qaxserver.def
```

在 .pro 文件中提到的 qaxserver.rc 和 qaxserver.def 这两个文件都是标准文件,可以从 Qt 的 src\activeqt\control 目录中把它们复制出来。

由 qmake 生成的 makefile 或者 Visual C++ 项目文件包含了把服务器注册到 Windows 注册表中的规则。要在最终用户的机器上注册服务器,可以使用 regsvr32 工具,可以在所有版本的 Windows 系统下使用该工具。

然后,我们可以在一个 HTML 页面中使用 <object> 标签来包含这个 Bouncer 组件:

```
<object id="AxBouncer"
        classid="clsid:5e2461aa-a3e8-4f7a-8b04-307459a4c08c">
<b>The ActiveX control is not available. Make sure you have built and
registered the component server.</b>
</object>
```

我们可以创建调用这些槽的各个按钮:

```
<input type="button" value="Start" onClick="AxBouncer.start()">
<input type="button" value="Stop" onClick="AxBouncer.stop()">
```

可以像使用其他任意的 ActiveX 控件一样来使用 JavaScript 或者 VBScript,由它们操作这个窗口部件。请参考本书例子中的 demo.html 文件,它是作为使用该 ActiveX 服务器的一个基本页面。

最后的例子是一个可以脚本化的 Address Book 应用程序。这个应用程序可以作为一个标准的 Qt/Windows 应用程序,或者也可以作为一个进程外的 ActiveX 服务器。后者可能允许我们脚本化这个应用程序,比方说使用 Visual Basic。

```
class AddressBook : public QMainWindow
{
    Q_OBJECT
    Q_PROPERTY(int count READ count)
    Q_CLASSINFO("ClassID", "{588141ef-110d-4beb-95ab-ee6a478b576d}")
    Q_CLASSINFO("InterfaceID", "{718780ec-b30c-4d88-83b3-79b3d9e78502}")
    Q_CLASSINFO("ToSuperClass", "AddressBook")

public:
    AddressBook(QWidget *parent = 0);
    ~AddressBook();

    int count() const;

public slots:
```

```
    ABItem *createEntry(const QString &contact);
    ABItem *findEntry(const QString &contact) const;
    ABItem *entryAt(int index) const;

private slots:
    void addEntry();
    void editEntry();
    void deleteEntry();

private:
    void createActions();
    void createMenus();

    QTreeWidget *treeWidget;
    QMenu *fileMenu;
    QMenu *editMenu;
    QAction *exitAction;
    QAction *addEntryAction;
    QAction *editEntryAction;
    QAction *deleteEntryAction;
};
```

AddressBook 窗口部件是这个应用程序的主窗口。它所提供的属性和槽可用于编写脚本语言。Q_CLASSINFO()宏用于确定这个类和与这个类相关的接口的 ID。通过使用像 guid 或者 uuid 这样的工具,这些都可以产生出来。

在前面的例子中,当使用QAXFACTORY_DEFAULT()宏导出 QAxBouncer 类的时候,我们给定了类和接口的 ID。在这个例子中,我们希望可以导出多个类,因而不能再使用QAXFACTORY_DEFAULT()宏。对我们来讲,还有两个选择可以使用:

- 可以子类化 QAxFactory,重新实现它的虚函数以提供我们想要导出的类型的相关信息,并且可以使用 QAXFACTORY_EXPORT()宏来注册其工厂方法。
- 可以使用宏 QAXFACTORY_BEGIN()、QAXFACTORY_END()、QAXCLASS()和 QAXTYPE()来声明和注册工厂。这一方法需要我们使用 Q_CLASSINFO()来指明类和接口的 ID。

回到 AddressBook 类的定义:Q_CLASSINFO()的第三次出现可能有些不同寻常。默认情况下,ActiveX 控件不仅会暴露它们自己的属性、信号和连接到客户端的槽,而且还会暴露它们向上直到 QWidget 的父对象类。ToSuperClass 属性可以指定我们想要在继承树中暴露的最高的父对象类。这里,指定该组件(AddressBook)的类名字来作为要导出的最高的父对象类,也就是说,在 AddressBook 的父对象类中继承的那些属性、信号以及槽将不会被导出。

```
class ABItem : public QObject, public QTreeWidgetItem
{
    Q_OBJECT
    Q_PROPERTY(QString contact READ contact WRITE setContact)
    Q_PROPERTY(QString address READ address WRITE setAddress)
    Q_PROPERTY(QString phoneNumber READ phoneNumber
               WRITE setPhoneNumber)
    Q_CLASSINFO("ClassID", "{bc82730e-5f39-4e5c-96be-461c2cd0d282}")
    Q_CLASSINFO("InterfaceID", "{c8bc1656-870e-48a9-9937-fbe1ceff8b2e}")
    Q_CLASSINFO("ToSuperClass", "ABItem")

public:
    ABItem(QTreeWidget *treeWidget);

    void setContact(const QString &contact);
    QString contact() const { return text(0); }
    void setAddress(const QString &address);
    QString address() const { return text(1); }
    void setPhoneNumber(const QString &number);
    QString phoneNumber() const { return text(2); }
```

```
public slots:
    void remove();
};
```

ABItem 类表示了地址簿中的一项。它从 QTreeWidgetItem 中派生而来，所以它可以显示在一个 QTreeWidget 中。同时，它也还从 QObject 中派生出来，因而可以导出为一个 COM 对象。

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!QAxFactory::isServer()) {
        AddressBook addressBook;
        addressBook.show();
        return app.exec();
    }
    return app.exec();
}
```

在 main()中，我们检查应用程序是将独立运行还是将用作服务器。-activex 命令行选项可以由 QApplication 识别，并且它可以让这个应用程序运行为服务器。如果该应用程序不是作为运行服务器，那么就可以像在任何一个单独运行的 Qt 应用程序中所做的那样来创建它的主窗口部件并且显示它。

除了-activex 选项以外，ActiveX 服务器还可以理解下面这些命令行选项：

- -regserver 会把这个服务器注册到系统注册表中。
- -unregserver 会从系统注册表中取消这个服务器的注册。
- -dumpidl *file*.*idl* 会把这个服务器的 IDL 输出到一个给定的文件中。

当应用程序作为服务器而运行时，必须把 AddressBook 和 ABItem 两个类作为 COM 组件导出：

```
QAXFACTORY_BEGIN("{2b2b6f3e-86cf-4c49-9df5-80483b47f17b}",
                 "{8e827b25-148b-4307-ba7d-23f275244818}")
QAXCLASS(AddressBook)
QAXTYPE(ABItem)
QAXFACTORY_END()
```

上面的这些宏导出一个创建 COM 对象的工厂。由于我们想导出两种类型的 COM 对象，所以不能像前一个例子中所做的那样只是简单地使用 QAXFACTORY_DEFAULT()。

QAXFACTORY_BEGIN() 的第一个参数是类型库 ID；第二参数是应用程序的 ID。在 QAXFACTORY_BEGIN()和 QAXFACTORY_END()之间，我们可以指定所有可以初始化的类以及所有我们想作为 COM 对象而访问的数据类型。

以下是用于进程外的 ActiveX 服务器的 .pro 文件：

```
TEMPLATE      = app
CONFIG       += qaxserver
HEADERS       = abitem.h \
                addressbook.h \
                editdialog.h
SOURCES       = abitem.cpp \
                addressbook.cpp \
                editdialog.cpp \
                main.cpp
FORMS         = editdialog.ui
RC_FILE       = qaxserver.rc
```

在 .pro 文件中提到的 qaxserver.rc 文件是一个标准文件，可以从 Qt 的 src\activeqt\control 目录中把它复制出来。

还可以在示例程序的 vb 目录里面查找一个使用 Address Book 服务器的 Visual Basic 项目。

到此为止,我们就完成了 ActiveQt 框架的回顾。Qt 的发行版中还包含了一些其他的例子,并且文档中也包含了有关如何构建 QAxContainer 和 QAxServer 的信息,以及如何解决常见的协同工作问题的一些信息。

23.3 处理 X11 会话管理

当从 X11 注销的时候,一些窗口管理器会询问我们是否希望保存会话。如果回答"是",那么在下一次登录进来时,这些应用程序就会自动恢复到重新运行的状态,而且它们会具有与我们注销时一样的屏幕位置。在理想情况下,它们的状态也可以保持与我们在注销时的状态一样。这方面的一个例子如图 23.7 所示。

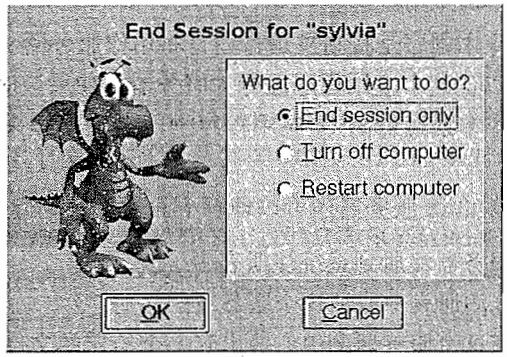


图 23.7 KDE 上的注销

用于注意保存和恢复会话的与 X11 相关的组件称为会话管理器(session manager)。为了让一个 Qt/X11 应用程序可以感知到这个会话管理器,必须重新实现 QApplication:: saveState(),并且把应用程序的状态保存在那里。

Microsoft Windows 和一些 UNIX 操作系统都提供了一种不同的机制,称为休眠(hibernation)。当用户让计算机进入休眠的时候,操作系统只是简单地把计算机的内存存放到磁盘中,并且在唤醒计算机的时候再重新加载它。但应用程序在系统休眠的时候不需要做任何事情,或者说它甚至根本就没有感觉到发生了系统休眠。

当用户启动一个关闭(shutdown)操作的时候,通过重新实现 QApplication::commitData(),就可以恰好在关闭发生之前得到控制权。这样就可以允许用户保存任何未保存的数据,并且在需要的情况下还可以和用户进行交互。会话管理中的这一部分内容在 X11 和 Windows 上都是支持的。

我们将通过仔细查看一个如图 23.8 所示的可以感知会话的 Tic-Tac-Toe 应用程序的代码来研究会话管理。首先,查看一下 main()函数的代码:

```
int main(int argc, char *argv[])
{
    Application app(argc, argv);
    TicTacToe toe;
    toe.setObjectName("toe");
    app.setTicTacToe(&toe);
    toe.show();
    return app.exec();
}
```

我们创建一个 Application 对象。这个 Application 类从 QApplication 类派生而来。同时,我们重新实现了 commitData()和 saveState()来支持会话管理。

接下来，我们创建一个 TicTacToe 窗口部件，让这个 Application 对象可以感知它，并且显示它。我们曾经把这个 TicTacToe 窗口部件称为“toe”。如果希望会话管理器可以恢复这个窗口的大小和位置，就必须给顶级窗口部件指定一个唯一的名字。

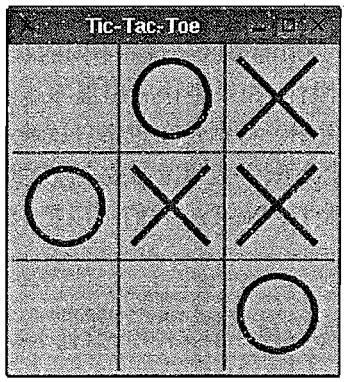


图 23.8　Tic-Tac-Toe 应用程序

以下是 Application 类的定义：

```
class Application : public QApplication
{
    Q_OBJECT

public:
    Application(int &argc, char *argv[]);

    void setTicTacToe(TicTacToe *toe);
    void saveState(QSessionManager &sessionManager);
    void commitData(QSessionManager &sessionManager);

private:
    TicTacToe *ticTacToe;
};
```

Application 类保存一个指向 TicTacToe 窗口部件的私有指针变量。

```
void Application::saveState(QSessionManager &sessionManager)
{
    QString fileName = ticTacToe->saveState();

    QStringList discardCommand;
    discardCommand << "rm" << fileName;
    sessionManager.setDiscardCommand(discardCommand);
}
```

在 X11 上，当会话管理器希望这个应用程序保存它自己的状态时，就会调用 saveState()函数。这个函数也可用于其他平台，但是它从来没有被调用过。QSessionManager 参数允许我们和会话管理器进行通信。

我们从要求 TicTacToe 窗口部件把它的状态保存到一个文件开始。然后，设置会话管理器的放弃命令(discard command)。放弃命令是这个会话管理器必须执行的，它用于删除关于当前状态的任何存储信息。对于这个例子，我们把它设置为：

```
rm sessionfile
```

这里的 *sessionfile* 就是含有会话中要保存状态的文件名，而 rm 是用于删除文件的标准 UNIX 命令。

会话管理器还有一个重新启动命令(restart command)。它是会话管理器必须执行的，用来重新启动应用程序。默认情况下，Qt 提供下面这样的重新启动命令：

```
appname -session id_key
```

第一部分的 *appname* 来自于 argv[0]。*id* 部分就是由会话管理器提供的会话 ID,需要确保它是唯一的,以用于区分不同的应用程序以及相同的应用程序的不同运行实例。*key* 部分会在唯一标识后添加一个用于表明状态保存时候的时间。由于各种各样的原因,会话管理器可能会在同一会话期间多次调用 saveState(),并且必须能够区分这些不同的状态。

因为现存会话管理器的一些局限性,如果想让应用程序能够正确地重新启动,就必须确保应用程序的路径存在于 PATH 环境变量中。特别是在想亲自试验一下 Tic-Tac-Toe 实例时,你就必须把它安装到(比方说)/usr/local/bin 目录下,并且以 tictactoe 的方式调用它。

对于那些简单的应用程序,包括 Tic-Tac-Toe,我们可以把保存状态作为重新启动命令的一个单独附加的命令行。例如:

```
tictactoe -state OX-XO-X-O
```

这样应当就可以不必再把数据保存到一个文件中,并且不需要再提供一条删除这个文件的放弃命令。

```
void Application::commitData(QSessionManager &sessionManager)
{
    if (ticTacToe->gameInProgress()
            && sessionManager.allowsInteraction()) {
        int r = QMessageBox::warning(ticTacToe, tr("Tic-Tac-Toe"),
                    tr("The game hasn't finished.\n"
                       "Do you really want to quit?"),
                    QMessageBox::Yes | QMessageBox::No);
        if (r == QMessageBox::Yes) {
            sessionManager.release();
        } else {
            sessionManager.cancel();
        }
    }
}
```

当用户注销的时候,就会调用 commitData()函数。我们可以重新实现它,让它可以弹出一个消息框,警告用户可能存在数据丢失的危险。默认实现会关闭所有的顶层窗口部件,这和用户一个接一个地单击窗口标题栏上的 close 按钮来关闭这些窗口的行为一样。在第 3 章中,我们已经看到了如何重新实现 closeEvent()来满足这一要求并且弹出一个消息框的示例。

为了达到这个例子的所要完成的目的,我们重新实现了 commitData(),并且如果还有任何一个游戏程序在运行的时候,我们就允许会话管理器能够和用户进行交互,弹出一个消息框来要求用户确认注销操作(如图 23.9 所示)。如果用户单击的是 Yes,就调用 release()来告诉会话管理器继续注销操作;如果用户单击的是 No,就调用 cancel()来取消注销操作。

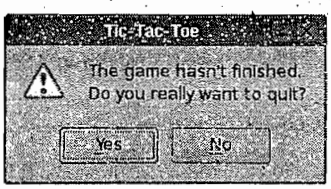


图 23.9 "Do you really want to quit?"

现在,让我们一起来看看 TicTacToe 类:

```
class TicTacToe : public QWidget
{
    Q_OBJECT
```

```
public:
    TicTacToe(QWidget *parent = 0);

    bool gameInProgress() const;
    QString saveState() const;
    QSize sizeHint() const;

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);

private:
    enum { Empty = '-', Cross = 'X', Nought = 'O' };

    void clearBoard();
    void restoreState();
    QString sessionFileName() const;
    QRect cellRect(int row, int column) const;
    int cellWidth() const { return width() / 3; }
    int cellHeight() const { return height() / 3; }
    bool threeInARow(int row1, int col1, int row3, int col3) const;
    char board[3][3];
    int turnNumber;
};
```

TicTacToe 类派生自 QWidget，并且重新实现了 sizeHint()、paintEvent()和 mousePressEvent()。它还提供了我们在 Application 类中使用过的 gameInProgess()和 saveState()两个函数。

```
TicTacToe::TicTacToe(QWidget *parent)
    : QWidget(parent)
{
    clearBoard();
    if (qApp->isSessionRestored())
        restoreState();

    setWindowTitle(tr("Tic-Tac-Toe"));
}
```

在构造函数中，会清空如图 23.8 所示的棋盘，并且如果应用程序是使用带有-session 选项调用时，就会调用私有函数 restoreState()来重新载入以前的状态。

```
void TicTacToe::clearBoard()
{
    for (int row = 0; row < 3; ++row) {
        for (int column = 0; column < 3; ++column) {
            board[row][column] = Empty;
        }
    }
    turnNumber = 0;
}
```

在 clearBoard()中，我们清空了所有单元格，并且把 trunNumber 设置为 0。

```
QString TicTacToe::saveState() const
{
    QFile file(sessionFileName());
    if (file.open(QIODevice::WriteOnly)) {
        QTextStream out(&file);
        for (int row = 0; row < 3; ++row) {
            for (int column = 0; column < 3; ++column)
                out << board[row][column];
        }
    }
    return file.fileName();
}
```

在 saveState()中，我们把棋盘的状态写到磁盘中。文件格式很简单，就是用“X”表示叉，用“O”表示圈，用“-”表示空白单元格。

```
QString TicTacToe::sessionFileName() const
{
    return QDir::homePath() + "/.tictactoe_" + qApp->sessionId() + "_"
           + qApp->sessionKey();
}
```

sessionFileName()私有函数会根据当前会话的 ID 和会话键(session key)返回其文件名。这个函数会由 saveState()和 restoreState()使用到。这个文件名可由会话 ID 和会话键得出。

```
void TicTacToe::restoreState()
{
    QFile file(sessionFileName());
    if (file.open(QIODevice::ReadOnly)) {
        QTextStream in(&file);
        for (int row = 0; row < 3; ++row) {
            for (int column = 0; column < 3; ++column) {
                in >> board[row][column];
                if (board[row][column] != Empty)
                    ++turnNumber;
            }
        }
    }
    update();
}
```

在 restoreState()中,我们载入与需要恢复的会话相关的文件,并且使用其中包含的信息填充棋盘。我们通过棋盘上的 X 和 O 的数量就可以得出 turnNumber 的值。

在 TicTacToe 构造函数中,如果 QApplication::isSessionRestored()返回 true,我们就调用 restoreState()。在这种情况下,sessionId()和 sessionKey()可以返回与保存应用程序状态时一致的值,这样 sessionFileName()就可以返回那个状态的文件名。

会话管理的测试和调试是一件令人倍感沮丧的事情,因为我们总是需要不停地登录和注销。要避免这种情况,一种可用的方法就是使用 X11 提供的标准 xsm 工具。在第一次调用 xsm 时,它会弹出一个会话管理器和一个终端。我们从这个终端中启动应用程序,这样就可以使用 xsm 作为它们的会话管理器,以此来替代那个通常的、系统范围内的会话管理器。然后,可以使用 xsm 的窗口来终止、重新启动或者放弃某一个会话,并且可以从中查看应用程序的行为是否和它预期的一致。有关如何做到这一点的详细信息,请参考 http://doc.trolltech.com/4.3/session.html。

第24章　嵌入式编程

开发那些运行在移动设备，比如PDA和手机上的软件是很有挑战性的，因为与桌面系统相比，嵌入式系统的处理器更慢，永久性存储容量（闪存或硬盘）更小，内存更少，并且显示器也更小。

Qt/Embedded Linux（也称Qtopia Core）是为嵌入式Linux优化过的Qt版本。Qt/Embedded Linux提供了Qt的桌面版（Qt/Windows、Qt/X11和Qt/Mac）相同的API和工具，并且加入了嵌入式编程所需的类和工具。通过双协议方式，可以在开源或者商业开发中使用它。

Qt/Embedded Linux可以在运行Linux的任何硬件上运行——包括Intel x86、MIPS、ARM、StrongARM、Motorola/Freescale 68000，以及PowerPC体系[①]。不像Qt/X11，它不需要X窗口系统，它实现的是自己的窗口系统，即QWS，从而大大节省了存储和内存。为了尽可能多地减少内存占用量，Qt/Embedded Linux可以被重新编译以去掉那些不用的特性。如果可以提前知道设备需要用到哪些应用程序或者组件，就可以将Qt/Embedded Linux的那些库静态地编译到一起。

Qt/Embedded Linux也得益于Qt桌面版的各种特性，包括把广泛应用的隐含数据共享（“写时复制”）作为节约内存的技术，利用QStyle支持客户化窗口部件风格，和一个可以高效利用屏幕空间的布局系统。

Qt/Embedded Linux构成了Trolltech嵌入式产品的基础，同时还包括Qtopia Platform、Qtopia PDA和Qtopia Phone。这些产品提供了专为移动设备设计的类和应用程序，可以与一些第三方Java虚拟机集成。

24.1　从Qt/Embedded Linux开始

Qt/Embedded Linux应用程序可以在任何安装了适当工具包的平台上开发。最通用的选择是在UNIX系统上创建GNU C++交叉编译环境。这一过程被Dan Kegel用一个脚本和一些补丁简化了，它们放在http://kegel.com/crosstool/上。在本章中，我们将使用Qtopia开源版4.2，可以从http://www.trolltech.com/products/qtopia/opensource/下载。这一版本只在Linux上可用，并且包含Qt/Embedded Linux 4.2的一个副本，还有一些支持Qtopia在桌面PC上编程的额外的工具。

Qt/Embedded Linux的配置系统支持交叉编译，这可以通过configure脚本的-embedded和-xplatform选项来实现。例如，为ARM体系做交叉编译需要键入：

```
./configure -embedded arm -xplatform qws/linux-arm-g++
```

可以通过向Qt的mkspecs/qws目录中添加新文件来创建用户自定义的配置脚本。

Qt/Embedded Linux直接在Linux的帧缓存中绘图（与视频显示相关联的内存区域）。如图24.1所示的虚拟帧缓存是一个X11应用程序，像素一一对应地模拟真正的帧缓存。要想访问帧缓存，可能需要拥有/dev/fb0设备的写操作权限。

要运行Qt/Embedded Linux应用程序，必须首先创建一个进程作为GUI服务器。服务器负责为客户端分配屏幕区域，生成鼠标和键盘事件。任何Qt/Embedded Linux应用程序都可以成为一个服

① 从4.4版开始，Qt有望运行在Windows CE上。

务器,需要在命令行指定-qws 参数或者将 QApplication::GuiServer 作为 QApplication 构造函数的第三个参数。

客户端应用程序通过共享内存和 UNIX 管道与 Qt/Embedded Linux 服务器通信。在后台,客户端将它们自己绘制到帧缓存中,并且负责绘制自己的窗口部件。

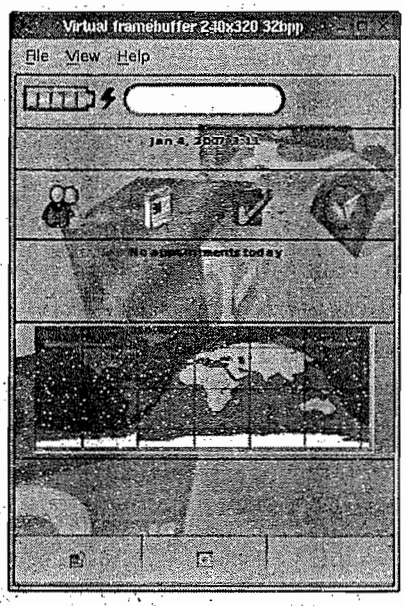


图 24.1 在虚拟帧缓存中运行 Qt/Embedded Linux

客户端可以通过 QCOP 彼此进行通信,这是一个 Qt 通信协议。一个客户端可以通过创建一个 QCopChannel 对象,连接它的 received()信号监听一个名字通道。例如:

```
QCopChannel *channel = new QCopChannel("System", this);
connect(channel, SIGNAL(received(const QString &, const QByteArray &)),
        this, SLOT(received(const QString &, const QByteArray &)));
```

QCOP 消息包含一个名字和一个可选的 QByteArray。静态函数 QCopChannel::send()在通道上广播消息。例如:

```
QByteArray data;
QDataStream out(&data, QIODevice::WriteOnly);
out << QDateTime::currentDateTime();

QCopChannel::send("System", "clockSkew(QDateTime)", data);
```

前面的例子阐述了一个理念:我们需要使用 QDataStream 编码,因为我们把这个数据格式放到了消息名称中,就如同它是一个 C++ 函数,确保接收者能够正确地解析 QByteArray。

各种各样的环境变量会影响到 Qt/Embedded Linux 应用程序,最重要的是 QWS_MOUSE_PROTO 和 QWS_KEYBOARD,它们用于指定鼠标设备和键盘类型。请参考 http://doc.trolltech.com/4.2/qtopiacore-envvars.html 上介绍的环境变量的完整列表。

如果将 UNIX 作为开发环境,可以使用 Qt 的虚拟帧缓存(qvfb)来测试应用程序,这是一个模拟实际帧缓存的 X11 应用程序。这在相当程度上加速了开发周期。为了在 Qt/Embedded Linux 上支持虚拟帧缓存,可以在配置脚本中加入-qvfb 参数。注意这个参数的目的不是做产品级的应用程序。虚拟帧缓存应用在 tools/qvfb 目录下,可以这样调用:

```
qvfb -width 320 -height 480 -depth 32
```

使用 X11 特性的虚拟帧缓存的另一个办法是使用虚拟网络计算(VNC, Virtual Network Computing)远程运行应用程序。为了使 Qt/Embedded Linux 支持 VNC,可以在配置中加入-qt-gfx-vnc 参数。然后,在命令行中加入-display VNC:0 来启动你的 Qt/嵌入式应用程序,并且运行一个 VNC 的客户端来指向你的应用程序运行的主机。显示的尺寸和位深可以在运行 Qt/Embedded Linux 应用的主机上通过设置 QWS_SIZE 和 QWS_DEPTH 环境变量来指定(例如, QWS_SIZE = 320 × 480, QWS_DEPTH = 32)。

24.2　自定义 Qt/Embedded Linux

在安装 Qt/Embedded Linux 时,可以通过指定一些特性来解决减少内存的占用问题。Qt/Embedded Linux 包含了 100 多个配置特性,每个特性都跟一个预定义的符号相关联。例如, QT_NO_FILEDIALOG 从 QtGui 库中去掉了 QFileDialog, Qt_NO_I18N 去掉了对国际化的全部支持。在 src/corelib/global/qfeatures.txt 中列出了这些特性。

Qt/Embedded Linux 提供了 5 种配置范例(minimum, small, medium, large 和 dist),它们保存在 src/corelib/global/qconfig-xxx.h 文件中。这些配置可以通过 configure 脚本的-qconfig *xxx* 参数指定。例如:

```
./configure -qconfig small
```

要创建用户的自定义配置,可以手动地提供一个 qconfig-*xxx*.h 文件,就像使用一个标准的配置文件一样来使用它。此外,还可以使用 qconfig 的图形化工具,它位于 Qt 的 tools 子目录中。

Qt/Embedded Linux 提供了下面这些类来负责与输入输出设备的交互,并且可以负责客户化窗口系统的外观。

类	作为基类的用途
QScreen	屏幕驱动
QScreenDriverPlugin	屏幕驱动插件
QWSMouseHandler	鼠标驱动
QMouseDriverPlugin	鼠标驱动插件
QWSKeyboardHandler	键盘驱动
QKdbDriverPlugin	键盘驱动插件
QWSInputMethod	输入法驱动
QDecoration	窗口外观装饰风格
QDecorationPlugin	窗口外观装饰风格插件

为了获得预定义驱动程序、输入法和窗口装饰风格的列表,可以在运行 configure 脚本的时候使用-help 选项。

在启动 Qt/Embedded Linux 服务器的时候,就像前一节中看到的那样,可以通过使用-display 命令行选项或者设置 QWS_DISPLAY 环境变量来指定屏幕驱动程序。可以通过设置 QWS_MOUSE_PROTO 环境变量来指定鼠标驱动程序和关联设备的驱动程序,这个环境变量的值必须使用 *type* : *device* 这样的语法,其中的 *type* 是所支持的驱动程序之一,而 *device* 是指向设备的路径(例如, QWS_MOUSE_PROTO = IntelliMouse:/dev/mouse)。使用类似的方式,通过指定 QWS_KEYBOARD 环境变量就可以处理键盘。可以在程序中使用 QWSServer::setCurrentInputMethod()和 QApplication::qwsSetDecoration()设定服务器的输入法和窗口装饰。

还可以通过对从 QStyle 继承的窗口部件风格的设置,自由地设置窗口的装饰风格。例如,设

置窗口装饰风格为 Windows 并且窗口部件风格为 Plastique 是完全可能的。只要你愿意,装饰可以基于每个窗口进行设置。

QWSServer 类提供了大量的自定义窗口系统的函数。运行在 Qt/Embedded Linux 服务器中的应用程序可以通过 QWSServer::instance()静态函数访问唯一的 QWSServer 实例。

Qt/Embedded Linux 支持如下字体格式:TrueType(TTF)、PostScript Type 1、位图发布格式(BDF),以及 Qt 预渲染字体(QPF)。

因为 QPF 是一种光栅格式的字体,所以它更加快速,并且如果只需要一种或两种字体大小的时候,它通常比 TTF 和 Type 1 这样的矢量字体还要小。makeqpf 工具可以让我们预渲染一个 TTF 或者 Type 1 文件并且把结果保存到 QPF 格式的文件中。另外一种方式就是使用-savefonts 命令行选项运行应用程序。

24.3 Qt 应用程序与 Qtopia 的集成

因为 Qt/Embedded Linux 提供了与 Qt 的桌面系统相同的 API,任何标准的 Qt 应用程序都可以被重新编译运行到 Qt/Embedded Linux 上。尽管如此,在实际应用中,为了解决屏幕较小、键盘有限(或者不存在键盘)以及资源受限地运行于 Qt/Embedded Linux 的微型设备上所出现的问题,创建专用应用程序是明智之举。此外,Qtopia 还提供了一些额外的库,用于支持 Qt/Embedded Linux 应用程序中使用到的移动设备的特性。

在使用 Qtopia 的 API 编写应用程序之前,必须编译和安装 Qtopia 的开发包,包括它自己的 Qt/Embedded Linux 的副本。这里假设使用的是 Qtopia 的开源版本 4.2,它几乎包含了 Qtopia 电话版的所有东西。

编译 Qtopia 不同于标准的 UNIX 应用程序,因为 Qtopia 不应当在源码目录下编译。例如,如果把 qtopia-opensource-src-4.2.4.tax.gz 源码包下载到$HOME/downloads 目录下,应该这样编译 Qtopia:

```
cd $HOME/downloads
gunzip qtopia-opensource-src-4.2.4.tar.gz
tar xvf qtopia-opensource-src-4.2.4.tar
```

现在,必须创建一个 Qtopia 的编译目录,例如:

```
cd $HOME
mkdir qtopia
```

为了方便,文档里建议设置 QPEDIR 环境变量。例如:

```
export QPEDIR=$HOME/qtopia
```

这里,假设使用的是 Bash 解释器。现在就可以编译 Qtopia 了:

```
cd $QPEDIR
$HOME/downloads/qtopia-opensource-src-4.2.4/configure
make
make install
```

我们没有指定任何的配置参数,但你可能希望这样做。运行 configure -help 可以看到所有可能用到的那些参数。

安装完成之后,所有的 Qtopia 文件将位于$QPEDIR/image 目录中,由用户创建的与 Qtopia 交互的文件将位于$QPEDIR/home 目录中。在 Qtopia 看来,一个"镜像"就是 Qtopia 的一个文件系统,它位于桌面计算机中,由 Qtopia 在虚拟帧缓存上运行时使用。

Qtopia 提供了复杂的文档集,并且熟悉它们是很重要的,因为 Qtopia 提供了许多 Qt 的桌面版无法使用(或者无关)的类。这些文档的起始页面是$QPEDIR/doc/html/index.html。

一旦编译和安装完成，就可以通过运行$ QPEDIR/scripts/runqtopia 来做一个测试。这个脚本调用带了皮肤的虚拟帧缓存以及 qpe，包含 Qtopia 应用程序栈的 Qtopia 环境。单独运行虚拟帧缓存以及 Qtopia 环境也是可以的，但需要以正确的顺序来启动它们。如果不小心首先启动了 qpe，Qtopia 就会写到 X11 的帧缓存，最好的结果也会破坏屏幕的显示。runqtopia 脚本可以通过使用-help 查看它所支持的命令行参数。这包括-skin，以及可选的皮肤列表。

虚拟帧缓存有一个上下文菜单，在 Qtopia 以外的区域单击右键就可以弹出。菜单有助于我们调整显示以及关闭 Qtopia。

既然 Qtopia 已经运行在虚拟帧缓存中，我们就可以编译一个自带的例子，看看进程是如何工作的。然后，我们创建一个非常简单的应用程序。

把目录切换到$ QPEDIR/examples/application 下。Qtopia 有它自己版本的 qmake，称为 qtopiamake，它位于$QPEDIR/bin 目录下。运行它来创建一个 makefile，然后运行 make。这将创建一个 example 可执行文件。然后，运行 make install 安装它，安装程序将会把 example(以及其他一些文件)复制到 Qtopia 的镜像文件夹中。现在，如果先关闭 Qtopia，然后再重新运行它，可以使用 runqtopia，我们就可以看到一个新的“Example”应用程序。要想运行这个应用程序，可以单击 Qtopia 底部中央的“Q”按钮，然后单击软件包的图标(“box”图标，手形标志的上方)，最后单击“Example”即可(如图 24.3所示)。

我们将通过创建一个很小但很有用的 Qtopia 应用程序来结束这一节，因为其中还有一些需要注意的细节。应用程序的名字称为单位换算(Unit Converter)，如图 24.2 所示。它只是使用了 Qt 的 API，因此没什么新奇的地方。下一节将使用一些 Qtopia 专有 API 来创建一个更复杂的例子。

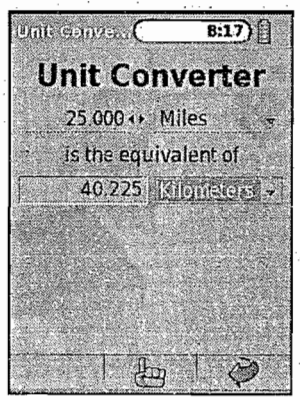


图 24.2　Unit Converter 应用程序

单位换算应用程序包含三个文件：main.cpp、unitconverter.h 和 unitconverter.cpp。创建一个新的 unitconverter 目录，然后进入该目录，创建一个空的 .cpp 和 .h 文件。然后运行：

```
qtopiamake -project
```

来创建一个 .pro 文件。这个文件类似于：

```
qtopia_project(qtopia app)

TARGET         = unitconverter
CONFIG        += qtopia_main no_quicklaunch
HEADERS       += unitconverter.h
SOURCES       += main.cpp \
                 unitconverter.cpp
pkg.domain     = none
```

即使我们写了代码,编译出了可执行文件,并且安装了它,它也不会出现在 Qtopia 的应用程序列表中。为了做到这一点,我们必须指定该应用程序图片的位置,即它的 .desktop 文件的位置,以及应该把它放在什么地方。提供一些如何打包方面的知识也是一个不错的练习。因此,我们来手动修改 .pro 文件,把它改成这样:

```
qtopia_project(qtopia app)

TARGET         = unitconverter
CONFIG        += qtopia_main no_quicklaunch
HEADERS       += unitconverter.h
SOURCES       += main.cpp \
                 unitconverter.cpp
INSTALLS      += desktop pics

desktop.files = unitconverter.desktop
desktop.path  = /apps/Applications
desktop.hint  = desktop

pics.files    = pics/*
pics.path     = /pics/unitconverter
pics.hint     = pics

pkg.name      = unitconverter
pkg.desc      = A program to convert between various units of measurement
pkg.version   = 1.0.0
pkg.domain    = window
```

现在的 .pro 文件包含一个 INSTALLS 选项,它的意思是说当运行 make install 时,应用程序的 .desktop 文件和图片必须跟可执行文件安装在一起。

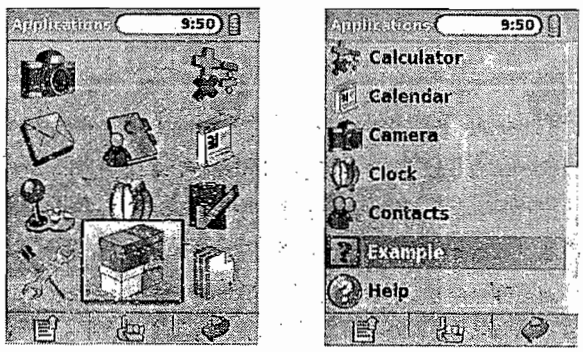


图 24.3 定位 Example 应用程序

根据惯例,图片会被安装在 pic 子文件夹中,.pro 文件中的 pics.*xxx* 项指定了原始图片的位置以及它们需要被安装到的位置。desktop.*xxx* 项指定应用程序的 .desktop 文件的名称以及它应当安装的位置。把它安装在/apps/Applications 目录下保证了当用户单击软件包图标时它可以出现在应用程序列表上。当应用程序在桌面环境中运行时,绝对路径,例如/apps/Applications 和/pics/expenses 实际上是 Qtopia 镜像目录中的一个相对路径(apps 会被 bin 所取代)。

unitconverter.desktop 文件提供了一些应用程序的基本信息。使用它的目的是可以在应用程序列表中显示它。这是完整的文件:

```
[Desktop Entry]
Comment[]=A program to convert between various units of measurement
Exec=unitconverter
Icon=unitconverter/Example
Type=Application
Name[]=Unit Converter
```

我们提供的这些信息只是能够指定的信息的子集。例如，可以提供翻译的信息。注意图标没有 .png 之类的扩展名。由 Qtopia 的资源系统去查找和显示合适的的图片。

现在，已经得到了一个特定的手动修改了的 .pro 文件以及一个 .desktop 文件。我们还需要做一件 Qtopia 特有的工作，然后就可以使用标准 Qt 语言并且按照其标准方式来编写 unitconverter.h 和 unitconverter.cpp 文件了。对于 Qtopia 来说，必须遵循一个特殊的习惯，与 qpe 关联起来。整个 main.cpp 则被裁减到只剩下这么几行：

```
#include <QtopiaApplication>

#include "unitconverter.h"

QTOPIA_ADD_APPLICATION("UnitConverter", UnitConverter)
QTOPIA_MAIN
```

主窗口类的名称是 UnitConverter，包含在 unitconverter.h 文件中。用来与 qtopiamake 和 unitconverter.h、unitconverter.cpp 文件关联，我们可以创建一个在 Qtopia 环境中运行的 Qtopia 应用程序。main()函数由 QTOPIA_MAIN 宏定义。

因为这个应用程序除了 main.cpp，使用的只是标准的 Qt 类，它也可以被编译成普通的 Qt 应用程序来运行。想要做到这一点，需要使用标准的 qmake，并且修改 main.cpp：

```
#include <QApplication>

#include "unitconverter.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    UnitConverter converter;
    converter.show();
    return app.exec();
}
```

此外，还需要注释掉 .pro 文件中的 qtopia_project()一行。

对于只需要在 Qt/Embedded Linux 下运行的应用程序来说，把它们如同标准的 Qt 应用程序一样开发通常是很方便的，或许可以显式地设定主窗口的 resize()为手机或者 PDA 的大小，当它们做完初版测试后再把它们修改为 Qtopia 应用程序。作为一种替代方法，可以有两个不同的 main.cpp 文件，或许是 main_desktop.cpp 和 main_qtopia.cpp，以及两个 .pro 文件。

单位转换应用程序中的大多数代码都与本书中介绍的其他 Qt 例子中的代码相类似，因此就不再逐一介绍了。

要想测试一个应用程序，必须运行 make，然后 make install，接着是 runqtopia。一旦 Qtopia 在虚拟帧缓存中运行起来，就可以单击“Q”按钮，然后单击软件包图标，接着单击“Unit Converter”。在这之后，就可以试验这个应用程序了，可以修改数量和选择组合框中的不同单位。

创建 Qtopia 应用程序与创建传统的 Qt 应用程序并没有太大的不同，只是除了对于初始化的设置以及用 qtopiamake 代替 qmake 等不同(特殊的 .pro 文件和 .desktop 文件)之外。测试嵌入式应用程序应该是很容易的，因为它们可以被编译、安装，还可以在虚拟帧缓存中运行。对于应用程序来说，尽管它们可以是传统的 Qt 应用程序，但实际上，把嵌入式环境的局限性考虑进去也是可取的，而且使用 Qtopia 特色的 API 可以确保它们跟 Qtopia 的应用程序栈很好地集成起来。

24.4　使用 Qtopia 的 API

Qtopia PDA 版和 Qtopia 电话版提供了一系列针对嵌入式用户的应用程序。大多数的这些应用

程序会被释放到库中,或者使用版本相关的库。这些库可以在 Qtopia 应用程序中使用,允许我们访问设备的服务,例如闹钟、邮件、打电话、SMS、语音录音,以及其他的一些应用程序等。

如果想在应用程序中访问设备相关的特性,可以有多种选择:

- 使用 Qt/Embedded Linux,编写自己的代码并与设备进行交互。
- 修改一个存在的 Qtopia 应用程序,使它具有所需的功能。
- 使用附加的 API 编写代码,例如,Qtopia 电话 API 或者 Qtopia PIM 应用程序的库。

本节将尝试最后一种方法。我们将编写一个小程序,记录简单的消费信息。它使用 Qtopia PIM 应用程序的数据来弹出一个联系人列表,然后向选定的联系人以 SMS 消息的形式发送消费报告。它还说明了如何使 Qtopia 支持大多数手机所具有的"软键盘"。

如图 24.4 所示,应用程序将位于应用程序软件包的列表中,就像前面一节中介绍的例子一样。跟以前一样,我们首先来看一下 .pro 文件,然后是 .desktop 文件,最后是应用程序源代码。这里是 expense 的 .pro 文件:

```
qtopia_project(qtopia app)
depends(libraries/qtopiapim)

CONFIG       += qtopia_main no_quicklaunch
HEADERS      += expense.h \
                expensedialog.h \
                expensewindow.h
SOURCES      += expense.cpp \
                expensedialog.cpp \
                expensewindow.cpp \
                main.cpp
INSTALLS     += desktop pics

desktop.files = expenses.desktop
desktop.path  = /apps/Applications
desktop.hint  = desktop

pics.files    = pics/*
pics.path     = /pics/expenses
pics.hint     = pics

pkg.name      = expenses
pkg.desc      = A program to record and SMS expenses
pkg.version   = 1.0.0
pkg.domain    = window
```

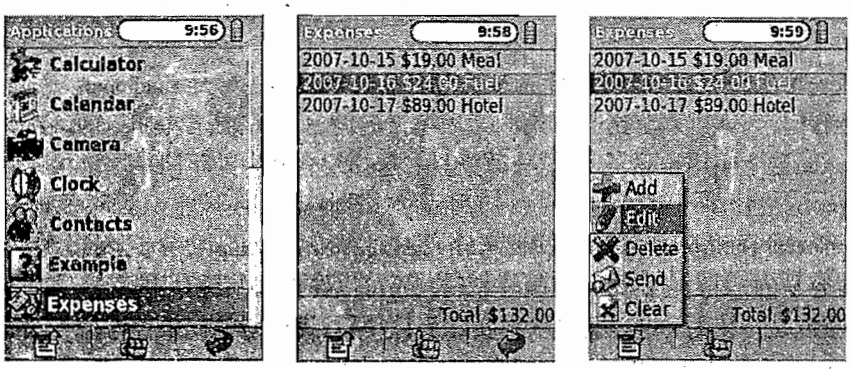


图 24.4 定位和运行 Expenses 应用程序

qtopia_project() 这一行跟前面一样。因为这个应用程序依赖于 Qtopia 的 PIM 库,我们使用 depends() 指定这个库。如果需要使用多个库,可以用逗号把它们隔开。余下的 .pro 文件跟我们在单位转换例子中看到的类似,只是这次有更多的源文件,因为这个应用程序更复杂一些。

expense.desktop 文件与前面看到的类似：

```
[Desktop Entry]
Comment[]=A program to record and SMS expenses
Exec=expenses
Icon=expenses/expenses
Type=Application
Name[]=Expenses
```

main.cpp 也一样：

```
#include <QtopiaApplication>

#include "expensewindow.h"

QTOPIA_ADD_APPLICATION("Expenses", ExpenseWindow)
QTOPIA_MAIN
```

现在介绍 Expenses 应用程序的头文件以及那些 Qtopia 特有或者与其显著相关的源码部分，首先从 Expense 类开始：

```
class Expense
{
public:
    Expense();
    Expense(const QDate &date, const QString &desc, qreal amount);

    bool isNull() const;
    void setDate(const QDate &date);
    QDate date() const;
    void setDescription(const QString &desc);
    QString description() const;
    void setAmount(qreal amount);
    qreal amount() const;

private:
    QDate myDate;
    QString myDesc;
    qreal myAmount;
};
```

这个简单的类保存了一个日期、一个描述信息以及一个账户。我们不介绍 expense.cpp 文件，因为其中没有一段代码是 Qtopia 所特有的，并且它也很简单。

```
class ExpenseWindow : public QMainWindow
{
    Q_OBJECT

public:
    ExpenseWindow(QWidget *parent = 0, Qt::WFlags flags = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void add();
    void edit();
    void del();
    void send();
    void clear();

private:
    void createActions();
    void createMenuOrToolBar();
    void loadData();
    void showData();
    ...
    QList<Expense> expenses;
};
```

ExpenseWindow 是应用程序的主窗体,它提供了一些方法,用户可以添加、编辑或删除个人消费信息,发送具有所有消费清单的 SMS 消息,并且可以清除这些信息。消费信息保存在 QList <Expense> 值中。

构造函数创建了一个 QListWidget 和两个 QLabel。一个标签显示文字"Total",另外一个显示消费的总和。这个动作由 createActions()函数产生,菜单或者工具栏由 createMenuOrToolBar()函数生成。两个函数都在构造函数中得到调用。所有已存在的记录都在构造函数的结尾处通过调用 loadData()函数加载进来。我们将跳过构造函数,而仅仅介绍一下它所调用的那些函数:

```
void ExpenseWindow::createActions()
{
    addAction = new QAction(tr("Add"), this);
    addAction->setIcon(QIcon(":icon/add"));
    connect(addAction, SIGNAL(triggered()), this, SLOT(add()));
    ...
    clearAction = new QAction(tr("Clear"), this);
    clearAction->setIcon(QIcon(":icon/clear"));
    connect(clearAction, SIGNAL(triggered()), this, SLOT(clear()));
}
```

createActions()函数创建了 Add、Edit、Delete、Send 以及 Clear 操作。尽管可以使用 Qt 的资源文件(.qrc),当编写 Qtopia 应用程序时,对于图标标准的做法是把它们保存在 pic 子目录中,然后复制到设备中(这要归功于 .pro 文件中的 INSTALLS 这一行)。为了在几个应用程序中都可以共享它们,Qtopia 使用一个特殊的数据库来优化对它们的访问。

在 Qt 或者 Qtopia 需要一个文件名的任何地方,都可以提供一个 Qtopia 资源的名称。通过文件名中打头的冒号来标识,紧跟着一个指定资源类型的单词。在这种情况下,我们指定需要的图标,给定一个文件名(例如,:icon/add),忽略文件的扩展名。Qtopia 会在一些标准的位置寻找一个合适的图标,首先会从应用程序的 pics 目录开始。详情可以查看 http://doc.trolltech.com/qtopia4.2/qtopia-resource-system.html。

```
void ExpenseWindow::createMenuOrToolBar()
{
#ifdef QTOPIA_PHONE
    QMenu *menuOrToolBar = QSoftMenuBar::menuFor(listWidget);
#else
    QToolBar *menuOrToolBar = new QToolBar;
    addToolBar(menuOrToolBar);
#endif

    menuOrToolBar->addAction(addAction);
    menuOrToolBar->addAction(editAction);
    menuOrToolBar->addAction(deleteAction);
    menuOrToolBar->addAction(sendAction);
    menuOrToolBar->addAction(clearAction);
}
```

有些电话有"软键盘",也就是说,具有一些多功能键,它们的操作是针对应用程序或者上下文环境的。QSoftMenuBar 类在合适的地方使用软键盘,否则弹出一个菜单。对于 PDA 来说,我们通常更希望把弹出菜单改为工具栏。# ifdef 标识确保在电话上使用菜单,而在 PDA 上使用工具栏。

用户可能希望能够在关闭应用程序时,不必显式地去保存数据。他们可能也希望稍后重新启动应用程序时数据能够得以恢复出来。这可以简单地通过调用构造函数中的 loadData()来实现,可以在应用程序的 closeEvent()中保存这些数据。Qtopia 提供了许多保存数据的方式,包括保存到 SQLite 数据库的表中或者保存到文件中。因为消费数据很小,所以将用 QSettings 来保存它。我们将看到是如何保存它的,以及是如何加载它的。

```
void ExpenseWindow::closeEvent(QCloseEvent *event)
{
    QByteArray data;

    QDataStream out(&data, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_2);

    foreach (Expense expense, expenses) {
        out << expense.date() << expense.description()
            << expense.amount();
    }

    QSettings settings("BookSoft Ltd", "Expenses");
    settings.setValue("data", data);

    event->accept();
}
```

我们创建一个 QByteArray,并且把所有的数据写进去。然后,在接收到关闭事件允许应用程序结束之前把它保存在 data 键值下。

```
void ExpenseWindow::loadData()
{
    QSettings settings("BookSoft Ltd", "Expenses");
    QByteArray data = settings.value("data").toByteArray();
    if (data.isEmpty())
        return;

    expenses.clear();
    QDataStream in(&data, QIODevice::ReadOnly);
    in.setVersion(QDataStream::Qt_4_2);

    while (!in.atEnd()) {
        QDate date;
        QString desc;
        qreal amount;

        in >> date >> desc >> amount;
        expenses.append(Expense(date, desc, amount));
    }
    showData();
}
```

数据来源于前面的会话,我们清空了存在的数据,然后读入每个新的消费项。showData()函数清空了列表窗口部件,然后循环加入消费信息,为每个消费信息添加一个新的列表项,然后更新标签的显示。

运行应用程序,用户可以添加、编辑或者删除消费项,用 SMS 消息发送它们,或者把它们全部清空。

对于删除操作,我们检查列表窗口部件中当前是否存在有效的行,然后使用一个标准的 QMessageBox::warning()静态便利函数询问用户确认删除操作。如果用户选择删除所有的消费信息,我们可以再使用一个消息框。所有这些操作都是标准的 Qt 程序。Qtopia 负责消息框的显示,以及与 Qtopia 环境的完全集成。

如果用户选择 Add 操作,则会添加一个新的消费项,从而会调用 add()槽:

```
void ExpenseWindow::add()
{
    ExpenseDialog dialog(Expense(), this);
    if (QtopiaApplication::execDialog(&dialog)) {
        expenses.append(dialog.expense());
        showData();
    }
}
```

这个槽创建了一个 ExpenseDialog,稍后将介绍这个类,但我们不是调用对话框的 QDialog::exec()函数,而是调用 QtopiaApplication::execDialog()函数,把对话框作为一个参数传进去。调用 exec()是可行的而且是有效的,但使用 execDialog()确保了对话框的大小和位置更适用于微型设备,如果有必要,它将被最大化。

Edit()槽与之类似。如果 edit()函数被调用,它会检查当前列表窗口部件中是否存在有效的行,如果有,它将该行对应的消费信息作为第一个参数传到 ExpenseDialog 的构造函数中。如果用户确认修改,修改后的信息将覆盖原有的信息。

最后一个需要介绍的 ExpenseWindow 函数是 send(),但在介绍它之前,我们将讨论一下 ExpenseDialog 类:

```
class ExpenseDialog : public QDialog
{
    Q_OBJECT

public:
    ExpenseDialog(const Expense &expense, QWidget *parent = 0);

    Expense expense() const { return currentExpense; }

public slots:
    void accept();

private:
    void createActions();
    void createMenuOrToolBar();

    Expense currentExpense;
    ...
};
```

很明显,我们有创建操作、菜单或者工具栏的函数,就如同 ExpenseWindow 类的定义。我们不会创建 QPushButton 或者 QDialogBuggonBox,而会创建工具栏或者 QSoftMenuBar,因为这些比创建按钮更容易与 Qtopia 环境进行集成。代码与我们创建应用程序的主窗口类似:

```
void ExpenseDialog::createActions()
{
    okAction = new QAction(tr("OK"), this);
    okAction->setIcon(QIcon(":icon/ok"));
    connect(okAction, SIGNAL(triggered()), this, SLOT(accept()));

    cancelAction = new QAction(tr("Cancel"), this);
    cancelAction->setIcon(QIcon(":icon/cancel"));
    connect(cancelAction, SIGNAL(triggered()), this, SLOT(reject()));
}
void ExpenseDialog::createMenuOrToolBar()
{
#ifdef QTOPIA_PHONE
    QMenu *menuOrToolBar = QSoftMenuBar::menuFor(this);
#else
    QToolBar *menuOrToolBar = new QToolBar;
    menuOrToolBar->setMovable(false);
    addToolBar(menuOrToolBar);
#endif

    menuOrToolBar->addAction(okAction);
    menuOrToolBar->addAction(cancelAction);
}
```

如果用户确认了操作,我们设置当前消费记录的日期、描述以及金额属性,并允许调用者使用对话框的 expense()函数检索数据。

如果用户选择了 Send 操作,send()函数就会得到调用。这个函数提示用户选择一个联系人来发送信息,准备要发送的文字,然后使用 SMS 协议发送该消息(如图 24.5 所示)。

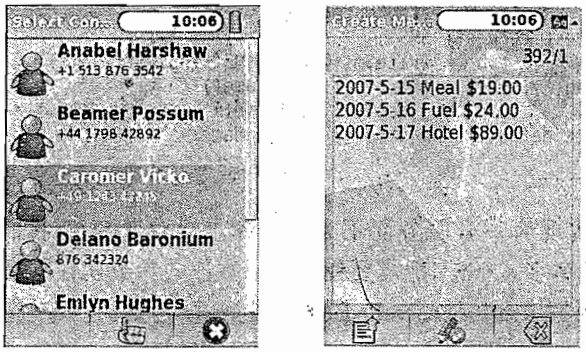


图 24.5　选择一个联系人并且发送 SMS 消息

```
void ExpenseWindow::send()
{
    QContactSelector dialog(false, this);
    dialog.setModel(new QContactModel);
    QtopiaApplication::execDialog(&dialog);
    if (!dialog.contactSelected())
        return;
```

QContactSelector 对话框和 QContactModel 模式/视图类都是由 PIM 库提供的。QContactModel 访问用户集中联系人信息的数据库。如果有很多的记录，QtopiaApplication::execDialog()将会最大化地弹出 QContactSelector 对话框。如果用户没有选择联系人，contactSelected()函数返回空联系人(意味着 false)，在这种情况下什么都不做。否则，准备发送消费信息：

```
QTemporaryFile file;
file.setAutoRemove(false);
if (!file.open()) {
    QMessageBox::warning(this, tr("Expenses"),
                         tr("Failed to send expenses: %1.")
                         .arg(file.errorString()));
    return;
}

QString fileName = file.fileName();
qreal total = 0.00;

QTextStream out(&file);
out.setCodec("UTF-8");

out << tr("Expenses\n");
foreach (Expense expense, expenses) {
    out << tr("%1 $%2 %3\n")
           .arg(expense.date().toString(Qt::ISODate))
           .arg(expense.amount(), 0, 'f', 2)
           .arg(expense.description());
    total += expense.amount();
}
out << tr("Total $%1\n").arg(total, 0, 'f', 2);
file.close();
```

为了发送 SMS 消息，我们需要传递保存 SMS 消息的文件名。这里，我们把消费数据用 QTextStream 保存到一个临时文件中。通常，QTemporaryFile 在调用 close()时删除文件，但我们关掉了这个操作，因为在 SMS 被发送之前文件必须存在，同时 Qtopia 将自动删除它。

total 变量是个 qreal 类型。这一类型是对 float 或者 double 的定义，依赖于体系结构。例如，在 ARM 上，因为性能问题，它代表一个 float 类型。在 Qt 的 API 中(特别是在 QPainter 中)，qreal 的使用胜过 double。

```
    QContact contact = dialog.selectedContact();
    QtopiaServiceRequest request("SMS",
                                 "writeSms(QString,QString,QString)");
    request << QString("%1 %2").arg(contact.firstName())
                               .arg(contact.lastName())
            << contact.defaultPhoneNumber() << fileName;
    request.send();
}
```

Qtopia 把 SMS 协议实现为一个服务而不是一个库。想发送 SMS,我们创建一个 QtopiaServiceRequest 对象,给它服务的名字,"SMS",把需要使用的函数连同参数列在圆括号中:"writeSms(QString,QString,QString)"。深入下去,QtopiaServicesRequest 使用 QCOP 与提供"SMS"服务的进程通信。

我们用接收者的名字和电话号码,以及文件的名字来创建一个请求,调用 send()把消息发出去。当 send()被执行,一个 Create Message 对话框会被 Qtopia 系统弹出来,文件中的消息的主体部分会显示在上面。用户可以修改这些文字,然后发送或者取消 SMS。Expense 应用程序只能在提供 SMS 服务的实际的或模拟的设备上进行测试。

如同本例中谈到的,嵌入式编程意味着我们必须考虑怎样使用和操作服务以及 Qtopia 特有的 API。它还要求我们以不同的思路考虑用户交互的设计,以解决嵌入式设备具有的微型屏幕和有限的输入设备的问题。从一个程序员的立场来看,除了必须熟悉 Qtopia 特有的那些额外的工具、库以及服务外,编写 Qtopia 应用程序跟编写桌面平台的应用程序并没有什么分别。

第四部分　附录

附录 A　Qt 的获取和安装

附录 B　编译 Qt 应用程序

附录 C　Qt Jambi 简介

附录 D　面向 Java 和C# 程序员的C++ 简介

附录 A　Qt 的获取和安装

本附录说明了如何获取并在你的系统中安装 GPL 版的 Qt。这些版本可用于 Windows、Mac OS X 和 X11(可适用于 Linux 和绝大多数的 UNIX)。可用于 Windows 和 Mac OS X 的预编译库中包含了 SQLite 和 SQLite 的驱动程序,SQLite 目前已用于公众数据库中。从源码包中编译而来的版本则可以自由选取是否包含 SQLite。开始之前,请先从 http://www.trolltech.com/download/下载 Qt 的最新版本。如果打算开发商业软件,那么将需要购买 Qt 的商业版,然后按照他们提供的安装说明进行安装即可。

Trolltech 还为像在 PDA 和移动电话这些基于 Linux 的嵌入式设备上构建应用程序提供了 Qt/Embedded Linux 版。如果对创建嵌入式应用程序感兴趣,那么可以从 Trolltech 的下载网页中下载 Qt/Embedded Linux。

用于本书的所有应用程序的例子都可以从本书的网站中下载到,网址是 http://www.informit.com/title/0132354160。另外,Qt 还提供了许多小示例应用程序,它们放在 examples 子目录中。

A.1　协议说明

一般说来,可以通过两种方式获取 Qt:一种是开源形式,另外一种是商业形式。开源版的各个版本都可以免费获取,而商业版的各个版本则需要通过一定的费用购买得到。

如果你希望发布自己基于 Qt 开源版创建的那些应用程序,那么就必须遵从在创建这个应用程序时所使用的 Qt 软件协议中列举出的那些特定条款和条件。对于开源版,这些条款和条件包括了使用 GNU 通用公共协议(GPL,General Public License)的要求。像 GPL 这样的开放协议会给予这个应用程序的用户一些特定的权利,包括查看和修改源代码以及重新发布这个应用程序(在同等条款下)的权利。如果希望在发布应用程序的同时不公布源代码(要保持源代码的私有性),或者如果希望在发布应用程序时使用自己的商业协议条件,那么就必须购买创建该应用程序时所使用到的那些商业版 Qt 软件。这些商业版允许基于自己的条款来销售和发布应用程序。

许可协议中的这些完整法律条款都包含在用于 Windows、Mac OS X 和 X11 的 GPL 版 Qt 中,其中也包含了如何获得商业版的信息。

A.2　Qt/Windows 的安装

在本书还在编写时,Qt 在 Windows 下的安装程序称为 qt-win-opensource-4.3.2-mingw.exe。这个版本号可能会与你阅读本书时所看到的版本号有所不同,但其安装过程则应当是相同的。要开始安装过程,请先把这个文件下载下来并运行它。

当安装程序运行到 MinGW 这一页时,如果你已经安装过 MinGW C++ 编译器,那么就必须指定它所在的目录;否则,请选中复选框,以便让安装程序自动为你安装 MinGW。GPL 版的 Qt 不能用于 Visual C++ ,所以,如果还没有安装 MinGW,那么就需要先安装它。安装程序会自动安装 Qt 的那些标准示例程序和参考文档。

如果安装时选择了安装 MinGW 编译器选项，那么在 MinGW 安装结束之前，可能还需要一段时间才能开始安装 Qt。

安装完成后，你将会在 Windows 的“开始”菜单中看到一个名为“Qt by Trolltech v4.3.2(Open-Source)”的新程序组。在这个程序组中，有指向 Qt 助手(Qt Assistant)和 Qt 设计师(Qt Designer)的快捷方式，还有一个名为 Qt 4.3.2 命令行提示符(Qt 4.3.2 Command Prompt)的快捷方式，它可以打开一个控制台窗口。打开这个窗口后，它能够自动设置使用 MinGW 编译器编译 Qt 程序时所需的环境变量。在这个窗口中，可以运行 qmake 和 make 命令来编译 Qt 应用程序。

A.3　Qt/Mac 的安装

在把 Qt 安装到 Mac OS X 之前，必须已经安装了 Apple 的 Xcode Tools 工具包。这些工具包通常会包含在和 Mac OS X 一起提供的那些 CD(或者 DVD)中，也可以从 Apple Developer Connection 中下载这些工具包，网址是 http://developer.apple.com。

如果使用的是 Mac OS X 10.4 和 Xcode Tools 2.x(with GCC 4.0.x)或者是之后版本，那么就可以使用如下描述的安装步骤；如果使用的是 Mac OS X 的早期版本，或者 GCC 是一个较早的版本，那么将需要手动安装这个源码包。该源码包的名称是 qt-mac-opensource-4.3.2.tar.gz，而且可以从 Trolltech 的网站中下载它。如果要安装这个软件包，那么请按照下一节中所给出的在 X11 上安装 Qt 的操作说明来进行相关安装。

要使用安装程序安装 Qt，请下载 qt-mac-opensource-4.3.2.dmg。(在写作本书时采用的就是这个版本，但当你阅读此书时，版本号可能已经与此有所不同了。)双击这个 .dmg 文件，然后再双击这个 Qt.mpkg 软件包。这样将会启动安装程序，它会把 Qt 的文档及其标准示例程序与 Qt 一起安装在/Developer 目录中。

要运行像 qmake 和 make 这样的命令，就需要使用一个终端窗口，例如，/Applications/Utilities 目录中的 Terminal.app。还有的情况就是要用 qmake 生成一个 Xcode 工程。例如，要为 hello 这个例子生成一个 Xcode 工程，就需要先打开一个控制台，比如 Terminal.app，然后把当前路径切换到 examples/chap01/hello，并且输入以下命令：

```
qmake -spec macx-xcode hello.pro
```

A.4　Qt/X11 的安装

从 Trolltech 的网站中下载文件 qt-x11-opensource-src-4.3.2.tar.gz。(在写作本书时采用的就是这个版本的文件，但当你阅读此书时，使用的文件可能已经发生了改变。)在 X11 中，要把 Qt 安装到它的默认位置，需要拥有 root 权限。如果没有 root 的访问权限，那么请使用 configure 工具的-prefix 选项来指定一个你具有写操作权限的目录。

1. 把当前路径切换到你存放下载文件的目录处。例如：

```
cd /tmp
```

2. 解压该压缩文件：

```
gunzip qt-x11-opensource-src-4.3.2.tar.gz
tar xvf qt-x11-opensource-src-4.3.2.tar
```

此时会生成一个/tmp/qt-x11-opensource-src-4.3.2 目录。Qt 需要的是 GNU 的 tar 工具，而在某些系统中它称为 gtar。

3. 用你喜欢的选项来执行 configure 工具,它可用于编译 Qt 库以及与 Qt 一起提供的工具软件:

```
cd /tmp/qt-x11-opensource-src-4.3.2
./configure
```

要查看 configure 的配置选项列表,可以运行 ./configure-help 命令。

4. 要编译 Qt,可以输入命令:

```
make
```

这样将会生成 Qt 库,同时也会编译所有的演示程序、示例程序和工具软件。在某些系统中,make 命令称为 gmake。

5. 要安装 Qt,可以输入命令:

```
su -c "make install"
```

然后输入 root 密码。(在某些系统中,上述命令是:sudo make install。)这样就可以把 Qt 安装到/usr/local/Trolltech/Qt-4.3.2 目录中。如果要改变安装路径,那么可以在 configure 命令的后面使用-prefix 选项来做到这一点。如果你已经对安装目录具有写操作权限的话,那么只需输入以下命令即可:

```
make install
```

6. 为 Qt 设置一些特定的环境变量。

如果使用的 shell 是 bash、ksh、zsh 或者 sh,那么请把以下两行代码添加到 .profile 文件中:

```
PATH=/usr/local/Trolltech/Qt-4.3.2/bin:$PATH
export PATH
```

如果使用的 shell 是 csh 或者 tcsh,那么请把下面一行代码添加到 .login 文件中:

```
setenv PATH /usr/local/Trolltech/Qt-4.3.2/bin:$PATH
```

如果使用了 configure 的-prefix 选项,那么请使用你自己指定的路径来代替这里给出的默认路径。

如果你正在使用的编译器不支持 rpath 命令,那么还必须扩展 LD_LIBRARY_PATH 环境变量,使其包含/usr/local/Trolltech/Qt-4.3.2/lib。对于带有 GCC 的 Linux 用户来讲,则没有必要执行这一步。

Qt 有一个演示程序 qtdemo,它可以充分地展示 Qt 库中的许多特色。也可以把它看成是开始查看 Qt 功能的完美工具。要查看 Qt 的文档,既可以访问 http://doc.trolltech.com 网站,也可以运行 Qt 助手,它是 Qt 的一个帮助程序,只需在控制台窗口中输入 assistant 命令即可把它调出来。

附录 B　编译 Qt 应用程序

编译工具的使用很大程度地简化了 Qt 应用程序的编译工作。我们可以使用三种方法来编译 Qt 应用程序：第一种方法是使用 Qt 提供的 qmake 工具，第二种方法是使用第三方的编译工具，而第三种方法是使用集成开发环境(IDE)。

qmake 工具可以使用与平台无关的 .pro 文件生成与平台相关的 makefile。该工具包含了调用 Qt 内置代码生成工具(moc、uic 和 rcc)的必要的逻辑规则。在本书的所有例子中都使用了 qmake，在多数情况下会使用相对简单的 .pro 文件。实际上，qmake 提供了很多的特性，包括创建可以递归调用其他 makefile 的 makefile，以及根据目前平台切换相关特性的开关状态等。本附录的第一节将会介绍 qmake，并且会介绍它的一些高级特性。

理论上，任何第三方的编译工具都可用于 Qt 的程序开发中，但使用可感知 Qt(Qt-aware)的工具会更容易些。第二节将介绍一些可感知 Qt 的编译工具。

一些开发者喜欢使用 IDE 编译应用程序。Trolltech 提供了可与 Visual Studio 和 Eclipse(如图 B.1 所示)相集成的软件，并且提供了一些开源的 IDE 软件，如 KDevelop 和 QDevelop，它们都是使用 Qt 编写的，并且为 Qt 开发提供了极好的支持。

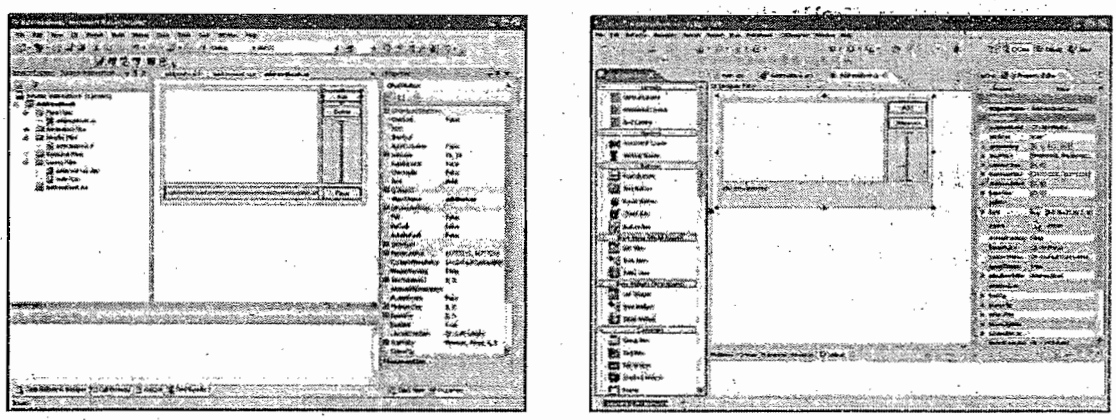


图 B.1　使用中的集成了 Qt 的 Visual Studio 和 Eclipse

B.1　使用 qmake

qmake 工具是与 Qt 一起提供的。它用来编译 Qt 本身，并且生成 Qt 自带的工具和例子。贯穿整书，我们一直使用 qmake 工程文件(.pro 文件)编译例子应用程序和插件。本节将系统(但非全面)地学习 .pro 文件的语法，并且会介绍几个 qmake 的基本概念。要想全面了解它们，请参阅 qmake 指南的在线帮助文档 http://doc.trolltech.com/4.3/qmake-manual.html。

.pro 文件的目的是列举工程中包含的源文件。由于 qmake 用于编译 Qt 及其相关工具，所以它很熟悉 Qt，并且能够生成一些触发 moc、uic 和 rcc 的规则。因此，qmake 的语法很简明，而且很容易学习。

工程文件主要分为三种：app(单独的应用程序)、lib(静态和动态库)和 subdirs(递归编译)。工

程文件的类型可以使用 TEMPLATE 变量指定如下:

```
TEMPLATE = lib
```

subdirs 模板可以用来编译子目录里的目标文件。在这种情况下,除 TEMPLATE = subdirs 外,还需要指定 SUBDIRS 变量。在每个子目录中,qmake 会搜寻以目录名命名的 .pro 文件,并且会编译该工程。用于本书例子中的 examples.pro 文件就使用了 subdirs 模板,可以分别在每个单独的例子上运行 qmake。

如果没有 TEMPLATE 这一项,那么默认工程是 app。对于 app 或者 lib 工程,最常使用的变量是下面这些:

- HEADERS 指定工程的 C++ 头文件(.h)。
- SOURCES 指定工程的 C++ 实现文件(.cpp)。
- FORMS 指定需要 uic 处理的由 Qt 设计师生成的 .ui 文件。
- RESOURCES 指定需要 rcc 处理的 .qrc 文件。
- DEFINES 指定预定义的 C++ 预处理器符号。
- INCLUDEPATH 指定 C++ 编译器搜索全局头文件的路径。
- LIBS 指定工程要链接的库。库既可以通过绝对路径指定,也可以使用源自 Unix 的-L 和-l 标识符来指定(例如,-L/usr/local/lib 和 -ldb_cxx)。
- CONFIG 指定各种用于工程配置和编译的参数。
- QT 指定工程所要使用的 Qt 模块(默认的是 core gui,对应于 QtCore 和 QtGui 模块)。
- VERSION 指定目标库的版本号。
- TARGET 指定可执行文件或库的基本文件名,其中不包含任何的扩展、前缀或版本号。(默认的是当前的目录名。)
- DESTDIR 指定可执行文件放置的目录。(默认值是平台相关的。例如,在 Linux 上,指当前目录;在 Windows 上,则是指 debug 或 release 子目录。)
- DLLDESTDIR 指定目标库文件放置的目录。(默认路径与 DESTDIR 相同。)

CONFIG 变量用来控制编译过程中的各个方面。它支持下面这些参数:

- debug 是指编译具有调试信息的可执行文件或者库,链接 Qt 库的调试版。
- release 是指编译不具有调试信息的可执行文件或者库,链接发行版的 Qt 库。如果同时指定 debug 和 release,则 debug 有效。
- warn_off 会关闭大量的警告。默认情况下,警告的状态是打开的。
- qt 是指应用程序或者库使用 Qt。这一选项是默认包括的。
- dll 是指动态编译库。
- staticlib 是指静态编译库。
- plugin 是指编译一个插件。插件总是动态库,因此这一参数暗指 dll 参数。
- console 是指应用程序需要写控制台(使用 cout,cerr,qWarning(),等等)。
- app_bundle 只适用于 Mac OS X 编译,是指可执行文件被放到束中,这是 Mac OS X 的默认情况。
- lib_bundle 只适用于 Mac OS X 编译,指库被放到框架中。

要生成工程文件 hello.pro 的 makefile,可以输入:

```
qmake hello.pro
```

在这之后,可以调用 make 或 nmake 编译工程。通过键入以下命令,也可以使用 qmake 生成一个 Microsoft Visual Studio 工程(.dsp 或 .vproj)文件:

```
qmake -tp vc hello.pro
```

在 Mac OS X 系统上，可以创建一个 Xcode 工程文件：

```
qmake -spec macx-xcode hello.pro
```

要创建 makefile，可以输入：

```
qmake -spec macx-g++ hello.pro
```

这里的-spec 命令行参数可以用来指定平台/编译器的组合。通常，qmake 可以正确地检测到所在的平台，但在某些情况下则有必要显式地指定平台情况。例如，在 linux 上以 64 位模式调用 Intel C++ 编译器(ICC)生成 makefile，应当输入：

```
qmake -spec linux-icc-64 hello.pro
```

那些可用的规则在 Qt 的 mkspecs 目录中。

尽管 qmake 的主要目的是生成 .pro 文件的 makefile，但也可以使用-project 参数在当前目录下使用 qmake 生成 .pro 文件，例如：

```
qmake -project
```

在这种模式下，qmake 将搜索当前目录下已知扩展名(.h、.cpp、.ui，等等)的文件，生成一个列举这些文件的 .pro 文件。

本节的余下部分将更详细地介绍 .pro 文件的语法。一个 .pro 文件中的条目的语法通常具有如下形式：

```
variable = values
```

values 是字符串的列表。注释以井号(#)开头，在行尾处结束。例如：

```
CONFIG = qt release warn_off        # I know what I'm doing
```

将列表["qt","release","warn_off"]赋给 CONFIG 变量，它会覆盖以前的各个值。额外的操作符作为 = 操作符的补充。+= 操作符可以用来扩展变量的值。因此：

```
CONFIG  = qt
CONFIG += release
CONFIG += warn_off
```

这几行会和前面的例子一样，可以有效地把列表["qt","release","warn_off"]赋值给 CONFIG 变量。-= 操作符从当前的变量中移除所有出现的指定的值。因此：

```
CONFIG  = qt release warn_off
CONFIG -= qt
```

会使 CONFIG 的值变成["release","warn_off"]。*= 操作在一个变量上添加一个值，但要求被添加的值不在变量的列表上；否则，就什么都不做。例如：

```
SOURCES *= main.cpp
```

这一行将把 main.cpp 实现文件添加到工程中，只有当还没有被添加的情况下才添加它。最后，~= 操作符使用指定的值替换符合正则表达式的值，这是一种类似于 sed(UNIX 流编辑器)的语法。例如：

```
SOURCES ~= s/\.cpp\b/.cxx/
```

使用 .cxx 替换 SOURCES 变量中所有 .cpp 文件的扩展名。

在值的列表中，qmake 提供了访问其他 qmake 变量、环境变量和配置参数的方法。图 B.2 列举了这些语法。

存取函数	说明
$$*varName* 或者$${*varName*}	.pro 文件中 qmake 变量在那一时刻的值
$$(varName)	当 qmake 运行时环境变量的值
$(varName)	当处理 makefile 时环境变量的值
$$[*varName*]	Qt 的配置参数值

图 B.2 qmake 的存取函数

前面的例子中使用的总是一些标准变量,例如 SOURCES 和 CONFIG,然而,我们也可以设置任意变量的值,并且可以使用$$*varName* 或者$${*varName*}语法引用它。例如:

```
MY_VERSION    = 1.2
SOURCES_BASIC = alphadialog.cpp \
                main.cpp \
                windowpanel.cpp
SOURCES_EXTRA = bezierextension.cpp \
                xplot.cpp
SOURCES       = $$SOURCES_BASIC \
                $$SOURCES_EXTRA
TARGET        = imgpro_$${MY_VERSION}
```

接下来的例子组合了前面介绍的几种语法,使用内置函数$$lower()把字符串转换为小写:

```
# List of classes in the project
MY_CLASSES    = Annotation \
                CityBlock \
                CityScape \
                CityView

# Append .cpp extension to lowercased class names, and add main.cpp
SOURCES       = $$lower($$MY_CLASSES)
SOURCES      ~= s/([a-z0-9_]+)/\1.cpp/
SOURCES      += main.cpp
# Append .h extension to lowercased class names
HEADERS       = $$lower($$MY_CLASSES)
HEADERS      ~= s/([a-z0-9_]+)/\1.h/
```

有时可能需要在 .pro 文件中指定包含空格的文件名。在这种情况下,只需简单地把文件名用引号括起来即可。

当在不同的平台上编译工程时,可能有必要基于平台指定不同的文件或者不同的参数。qmake 的条件判断语法是:

```
condition {
    then-case
} else {
    else-case
}
```

condition 部分可以是平台名字(例如,win32、unix 或者 macx),或者更复杂的断言。*then-case* 和 *else-case* 部分使用标准语法为变量赋值。例如:

```
win32 {
    SOURCES += serial_win.cpp
} else {
    SOURCES += serial_unix.cpp
}
```

else 分支是可选的。为了方便,当 *then-case* 部分仅有一条变量赋值,而且在没有 *else-case* 分支时,qmake 也支持单行形式的语法:

```
condition:then-case
```

例如：

```
macx:SOURCES += serial_mac.cpp
```

如果有几个工程文件需要共享相同的项，则可以把相同的项提取到单独的文件中，在各自的 .pro 文件中使用 include()语句包含它们：

```
include(../common.pri)

HEADERS      += window.h
SOURCES      += main.cpp \
                window.cpp
```

通常，打算被别的工程文件所包含的工程文件会带有 .pri(工程包含)的扩展名。

在前面的例子中，我们了解了$$lower()内置函数，它可以返回参数的小写版本。另外一个有用的函数是$$system()，它允许我们从外部应用程序中产生字符串。例如，如果想要确认当前使用的 UNIX 版本，可以这样写：

```
OS_VERSION = $$system(uname -r).
```

然后，可以在条件中使用结果变量，并与 contains()合用：

```
contains(OS_VERSION, SunOS):SOURCES += mythread_sun.c
```

本节只讲了一些表面的东西。qmake 工具提供了许多参数和特性，远多于这里所讲到的这些，包括对预编译头文件的支持、对 Mac OS X 通用二进制库的支持以及对用户定义的编译器或者其他工具的支持等。对于这方面更多信息的了解，可以参考 qmake 指南的在线帮助文档。

B.2 使用第三方编译工具

许多编译工具都可以用于编译 Qt 应用程序，包括开源的和商用的。这些工具可以分成两大类：一类是生成一些 makefile(或者一些 IDE 工程文件)并且依赖于标准编译系统的工具；另一类是不需要外部编译工具的单独的编译系统工具。

本节将介绍三种工具，它们都可以对 Qt 提供内置支持，或者是比较容易使用的。第一个工具是 CMake，它可以生成一些 makefile；另外两个分别是 Boost.Build 和 Scons，它们则是单独的编译系统。对于这里给出的每个工具系统，我们将介绍如何编译第 3 章和第 4 章中开发的 Spreadsheet 应用程序。如果要评估任何新的编译工具或者编译系统，都需要详细地研读并且对实际的应用程序进行实验，但我们希望在此给出的简短介绍能够说明这一问题。

Cmake：跨平台的 Make

CMake 工具是一个开源的跨平台 makefile 生成器，它内置了对 Qt 4 程序开发的支持，可以从网站 http://www.cmake.org/中获取它。为了使用 CMake，必须为工程创建一个 CMakeLists.txt 文件来描述它，该文件很像是一个 qmake.pro 文件。下面是为 Spreadsheet 应用程序写的 CMakeLists.txt 文件：

```
project(spreadsheet)
cmake_minimum_required(VERSION 2.4.0)
find_package(Qt4 REQUIRED)
include(${QT_USE_FILE})
set(spreadsheet_SRCS
    cell.cpp
    finddialog.cpp
    gotocelldialog.cpp
    main.cpp
    mainwindow.cpp
```

```
    sortdialog.cpp
    spreadsheet.cpp
)
set(spreadsheet_MOC_SRCS
    finddialog.h
    gotocelldialog.h
    mainwindow.h
    sortdialog.h
    spreadsheet.h
)
set(spreadsheet_UIS
    gotocelldialog.ui
    sortdialog.ui
)
set(spreadsheet_RCCS
    spreadsheet.qrc
)
qt4_wrap_cpp(spreadsheet_MOCS ${spreadsheet_MOC_SRCS})
qt4_wrap_ui(spreadsheet_UIS_H ${spreadsheet_UIS})
qt4_wrap_cpp(spreadsheet_MOC_UI ${spreadsheet_UIS_H})
qt4_add_resources(spreadsheet_RCC_SRCS ${spreadsheet_RCCS})
add_definitions(-DQT_NO_DEBUG)
add_executable(spreadsheet
    ${spreadsheet_SRCS}
    ${spreadsheet_MOCS}
    ${spreadsheet_MOC_UI}
    ${spreadsheet_RCC_SRCS})
target_link_libraries(spreadsheet ${QT_LIBRARIES} pthread)
```

文件的大部分都是简单的模板文件。我们必须填写的那些与应用程序相关的项目就是应用程序的名字(在第一行)、.cpp 源文件的列表、.ui 文件的列表,以及 .qrc 文件的列表。对于头文件,CMake 会足够聪明,它能够自己判断出它们之间的依赖关系,因此不需要指定。然而,由于用于定义 Q_OBJECT 类的 .h 文件需要被 moc 处理,所以必须列举这些文件。

开始的几行会设置应用程序的名字,为 Qt 4 添加 CMake 的支持,然后设置一些变量保存文件名。qt4_wrap_cpp() 命令会以给定的文件来运行 moc,同样,qt4_wrap_ui() 会运行 uic,而 qt4_add_resources() 则可以运行 rcc。为了创建可执行文件,需要指定所有的 .cpp 文件,包括由 moc 和 rcc 所生成的这些文件。最后,必须指定可执行文件要链接的库,这里会指定那些标准的 Qt 4 的库和线程库。

CMakeLists.txt 完成以后,就可以使用下面的命令生成 makefile:

```
cmake .
```

该命令会告诉 CMake 读取当前目录下的 CMakeLists.txt 文件并生成 Makefile 文件。然后,可以使用 make(或者 nmake)来编译应用程序,并且如果想从头开始编译,那么只需执行 make clean 即可。

Boost.Build(bjam)

Boost C++ 类库包含它们自己的编译工具,称为 Boost.Build 或者 bjam,其免费文档位于 http://www.boost.org/tools/build/v2/。该工具的第 2 版提供对 Qt 4 应用程序的内置支持,但前提是要有一个 QTDIR 的环境变量,以用来指定 Qt 4 的安装路径。有些 Boost.Build 的安装版在默认情况下是不支持 Qt 的。对于这些版本,必须编辑 user-config.jam,添加如下一行代码:

```
using qt ;
```

除了依赖 QTDIR 之外,还可以这样修改前面的那行代码来指定 Qt 的安装路径:

```
using qt : /home/kathy/opt/qt432 ;
```

使用 Boost.Build 编译的任何应用程序都需要两个文件:boost-build.jam 和 Jamroot。事实上,对

于任意多的应用程序我们只需要再复制 boost-build.jam 一次，并且只要把它放在包含所有应用程序目录的那个目录下(无论嵌套有多深)即可。boost-build.jam 文件只需要包含一行指定编译系统安装路径的代码即可。例如：

```
boost-build /home/kathy/opt/boost-build ;
```

用来编译 Spreadsheet 应用程序的 Jamroot 文件如下所示：

```
using qt : /home/kathy/opt/qt432 ;

exe spreadsheet :
    cell.cpp
    finddialog.cpp
    finddialog.h
    gotocelldialog.cpp
    gotocelldialog.h
    gotocelldialog.ui
    main.cpp
    mainwindow.cpp
    mainwindow.h
    sortdialog.cpp
    sortdialog.h
    sortdialog.ui
    spreadsheet.cpp
    spreadsheet.h
    spreadsheet.qrc
    /qt//QtGui
    /qt//QtCore ;
```

其中的第一行加入了对 Qt 4 的支持，我们必须在此提供 Qt 4 的安装路径。接着，指定可执行文件的名字为 spreadsheet，并且列出那些依赖文件。对于头文件，Boost.Build 足以推断出它们之间的依赖关系，因此通常不需要再列举它们。然而，由于必须使用 moc 处理那些定义了 Q_OBJECT 类的 .h 文件，所以必须列举它们。最后两行指定了想要使用的 Qt 库。

假设 bjam 可执行文件位于当前路径下，可以使用下面的命令编译应用程序：

```
bjam release
```

它会告知 Boost.Build 以发行版的方式编译当前目录下 Jamroot 文件所指定的应用程序。(如果只安装了 Qt 的调试版，则会产生错误，这种情况下可以运行 bjam debug。)在必要的情况下，可以运行 moc、uic 和 rcc 工具。想要清除结果，可以运行 bjam clean release。

SCons：软件构建工具

SCons 工具是一个基于 Python 的可以替换 make 的编译工具，其网址是 http://www.scons.org/。它可以对 Qt 3 提供内置支持，还有一个支持 Qt 4 的插件，它由 David García Garzón 提供，网址是 http://www.iua.upf.edu/~dgarcia/Codders/sconstools.html。从该网站可以下载一个文件 qt4.py，应该把它放到与应用程序相同的目录中。在未来适当的时候，该扩展将会包含到官方的 SCons 发行版中。

一旦把 qt4.py 放到了适当的位置，就可以创建一个 SConstruct 文件来指定编译条件：

```
#!/usr/bin/env python

import os

QT4_PY_PATH = "."
QTDIR = "/home/kathy/opt/qt432"

pkgpath = os.environ.get("PKG_CONFIG_PATH", "")
pkgpath += ":%s/lib/pkgconfig" % QTDIR
os.environ["PKG_CONFIG_PATH"] = pkgpath
os.environ["QTDIR"] = QTDIR
```

```
env = Environment(tools=["default", "qt4"], toolpath=[QT4_PY_PATH])
env["CXXFILESUFFIX"] = ".cpp"

env.EnableQt4Modules(["QtGui", "QtCore"])
rccs = [env.Qrc("spreadsheet.qrc", QT4_QRCFLAGS="-name spreadsheet")]
uis = [env.Uic4(ui) for ui in ["gotocelldialog.ui", "sortdialog.ui"]]
sources = [
    "cell.cpp",
    "finddialog.cpp",
    "gotocelldialog.cpp",
    "main.cpp",
    "mainwindow.cpp",
    "sortdialog.cpp",
    "spreadsheet.cpp"]
env.Program(target="spreadsheet", source=[rccs, sources])
```

这个文件是通过 Python 写出来的,因此我们可以使用 Python 语言的所有特性和库。

文件的大部分都是样本文件,只有几个与应用程序相关的项。首先设置 qt4.py 和 Qt 4 的安装路径。通过把 qt4.py 放在一个标准的位置,并设置相应的路径,就可以避免把它复制到每个应用程序目录中。我们必须显式地启用 Qt 模块,这里是 QtCore 和 QtGui,必须指定需要 rcc 或 uic 处理的文件。最后,我们列举源文件,设置程序名字,以及它依赖的源文件和资源文件。我们不需要指定 .h 文件,Qt 4 足以通过检查 .cpp 文件正确地运行 moc。

现在,可以使用 scons 命令来编译该应用程序:

```
scons
```

它将编译当前目录下 SConstruct 文件所指定的那个应用程序。也可以使用 scons-c 来清除结果。

附录 C　Qt Jambi 简介

Qt Jambi 是 Java 版的 Qt 应用程序开发框架。Qt Jambi 的核心部分是 Qt 的 C++ 库，Java 程序员可以通过 Java 的本地接口(Java Native Interface，JNI)使用它。尽管在把 Qt Jambi 集成到 Java 中作了许多努力，以使它的 API 可以被 Java 程序员正常使用，但 C++ /Qt 程序员仍然可以感觉得到这些 API 是熟悉而且是可以预见的。所有类的文档都使用 Javadoc 生成，请参阅 http://doc.trolltech.com/qtjambi/。

目前，Java GUI 程序员不得不使用 AWT、Swing、SWT 和类似的 GUI 类库，但它们都不如 Qt 方便和强大。例如，在传统的 Java GUI 库中，需要把用户的操作，例如点击按钮，连接到相应的具有事件监听类的方法中；而在 Qt Jambi 中，只需要一行代码就可以完成同样的事情。而且 Qt 的布局管理比 Swing 的 BoxLayout 和 GridBagLayout 要更好用些，并且可以生成更好的外观。

Qt Jambi 应用程序使用具有菜单条、工具栏、停靠窗口、状态栏的主窗口，就像使用 C++ 编写的 Qt 应用程序。它们还具有与平台相关的外观，并且兼顾了用户对主题、颜色、字体等的偏好。在 Qt 的鼎力支持下，Qt Jambi 应用程序可以使用 Qt 强大的二维图形体系(特别是图形化的视图框架)及其扩展，例如 OpenGL。

Qt Jambi 的好处不只局限于 Java 程序员。特别是 C++ 程序员可以使用程序生成器工具把他们自定义的 Qt 组件转换成 Java 程序员同样可以使用的组件，该程序生成器工具与由 Trolltech 提供的用于把 Qt 的 API 转换成适用于 Qt Jambi 中 API 的那个工具相同。

本附录将介绍 Java 程序员是如何使用 Qt Jambi 来创建图形用户界面应用程序的。然后，将介绍如何在集成了 Qt 设计师的 Eclipse 中使用 Qt Jambi，最后介绍如何让 Qt Jambi 程序员使用那些自定义的 C++ 组件。本附录将假定你很熟悉如何使用 C++ /Qt 和 Java 进行编程。而 Qt Jambi 则需要使用 1.5 版或者更高版本的 Java。

C.1　Qt Jambi 入门

在这一节中，我们将开发一个小的 Java 应用程序，用于显示如图 C.1 所示的窗口。除了窗口的名字外，Jambi Find 对话框与第 2 章创建的 Find 对话框有相同的外观和行为。通过使用相同的例子，可以更容易地了解 C++ /Qt 和 Qt Jambi 程序开发的不同点和相同点。在阅读代码的同时，我们将讨论 C++ 和 Java 在概念上的不同。

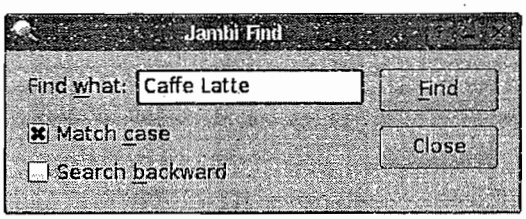


图 C.1　Jambi Find 对话框

Jambi Find 应用程序的实现放在一个称为 FindDialog.java 的文件中。我们将分段介绍该文件

的内容,首先从 import 的声明开始。

```
import com.trolltech.qt.core.*;
import com.trolltech.qt.gui.*;
```

在这两行的作用下,两个 import 声明使 Java 可以使用 Qt 的 core 和 GUI 类。同样可以使用类似的 import 声明方式导入其他的那些类(例如,import com.trolltech.qt.opengl.*)。

```
public class FindDialog extends QDialog {
```

FindDialog 类是 QDialog 的一个子类,就像在 C++ 版中所做的那样。C++ 在头文件中声明信号,依靠 moc 工具生成支持的代码。在 Qt Jambi 中,Java 的内省技术用来实现信号和槽机制。但仍然需要一些声明信号的工具,这可以通过一些 Signal*N* 类来实现:

```
public Signal2<String, Qt.CaseSensitivity> findNext =
        new Signal2<String, Qt.CaseSensitivity>();

public Signal2<String, Qt.CaseSensitivity> findPrevious =
        new Signal2<String, Qt.CaseSensitivity>();
```

一共有 10 个 Signal*N* 类——Signal0,Signal1<T1>,…,Signal9<T1,…,T9>。名字中的数字代表它们带的参数数量,类型 T1,…,T9 用于指定参数类型。这里声明了两个信号,每个信号都带两个参数。在这两种情况下,第一个参数是 Java String,第二个参数是 Qt.CaseSensitivity,这是一个 Java 型的枚举。在 Qt API 中使用 QString 的地方,在 Qt Jambi 中则使用 String 来代替。

不像其他的那些 Signal*N* 类,Signal0 不是一个泛型类。为了创建一个没有参数的信号,可以这样使用 Signal0:

```
public Signal0 somethingHappened = new Signal0();
```

创建完信号,就可以开始介绍构造函数的实现了。函数很长,因此分成三部分来介绍。

```
public FindDialog(QWidget parent) {
    super(parent);

    label = new QLabel(tr("Find &what:"));
    lineEdit = new QLineEdit();
    label.setBuddy(lineEdit);

    caseCheckBox = new QCheckBox(tr("Match &case"));
    backwardCheckBox = new QCheckBox(tr("Search &backward"));

    findButton = new QPushButton(tr("&Find"));
    findButton.setDefault(true);
    findButton.setEnabled(false);

    closeButton = new QPushButton(tr("Close"));
```

在 Java 和 C++ 中创建窗口部件的唯一不同点就是语法的细节稍有不同。注意 tr()会返回一个 String,而不是一个 QString。

```
lineEdit.textChanged.connect(this, "enableFindButton(String)");
findButton.clicked.connect(this, "findClicked()");
closeButton.clicked.connect(this, "reject()");
```

在 Qt Jambi 中,信号-槽的连接语法跟 C++/Qt 稍有不同,但仍然很简洁。一般情况下,其语法是:

```
sender.signalName.connect(receiver, "slotName(T1, ..., TN)");
```

不像 C++/Qt 那样,我们不需要给信号指定签名。如果信号的参数多于它连接的槽的参数,多余的参数都会被忽略掉。此外,在 Qt Jambi 中,信号-槽机制不仅局限于 QObject 子类,任何继承于 QSignalEmitter 的类都可以发送信号,而任意类的任意方法都可以作为槽函数。

```
        QTransform transform;
        transform.translate(+50.0, +50.0);
        transform.rotate(+45.0);
        transform.translate(-50.0, -50.0);
        painter.setWorldTransform(transform);
        painter.drawText(pos, tr("Sales"));

        QVBoxLayout rightLayout = new QVBoxLayout();
        rightLayout.addWidget(findButton);
        rightLayout.addWidget(closeButton);
        rightLayout.addStretch();

        QHBoxLayout mainLayout = new QHBoxLayout();
        mainLayout.addLayout(leftLayout);
        mainLayout.addLayout(rightLayout);
        setLayout(mainLayout);

        setWindowTitle(tr("Jambi Find"));
        setFixedHeight(sizeHint().height());
    }
```

这些用于布局的代码几乎就是原来 C++ 中的原有代码，因而具有相同的布局和相同的效果。在 Qt Jambi 中，也可以使用 Qt 设计师创建窗体，也会使用到 juic(用于 Java 用户接口的编译器)，我们将在下一节予以介绍。

```
    private void findClicked() {
        String text = lineEdit.text();
        Qt.CaseSensitivity cs = caseCheckBox.isChecked()
                ? Qt.CaseSensitivity.CaseSensitive
                : Qt.CaseSensitivity.CaseInsensitive;
        if (backwardCheckBox.isChecked()) {
            findPrevious.emit(text, cs);
        } else {
            findNext.emit(text, cs);
        }
    }
```

Java 型枚举的语法比 C++ 更详细些，但是很容易理解。要想发送一个信号，可以使用 Signal*N* 对象调用 emit()函数，并传递一些正确类型的参数。在程序编译结束后，类型检查就完成了。

```
    private void enableFindButton(String text) {
        findButton.setEnabled(text.length() == 0);
    }
```

enableFindButton()函数本质上与 C++ 的源代码相同。

```
    private QLabel label;
    private QLineEdit lineEdit;
    private QCheckBox caseCheckBox;
    private QCheckBox backwardCheckBox;
    private QPushButton findButton;
    private QPushButton closeButton;
```

为了与本书其他的代码保持一致，我们声明所有的窗口部件都作为类的私有域。这纯粹是个风格上的问题，没有什么能够阻止我们在构造函数中声明那些只在构造函数中使用的窗口部件。例如，可以在构造函数中声明 label 和 closeButton，因为它们不被别的地方引用，而且它们也不需要构造后就进行垃圾信息回收。因为 Qt Jambi 使用和 C++ /Qt 相同的父-子类机制，因此一旦 label 和 closeButton 创建成功，FindDialog 窗体就是其所有者，在程序后台，它将保持对它们的引用，使其不被销毁。Qt Jambi 递归删除子窗口部件，因此如果上层窗口被删除了，该窗口将逐个删除其子窗口和布局，以及它们的子窗口部件，直到整个清理过程完成为止。

使用 Java 的资源系统

Qt Jambi 十分清楚 Java 的资源系统,不像 Java 的许多标准类。Java 资源由 classpath:前缀定义。任何在 Qt Jambi API 中使用的文件名,都可以用 Java 的资源进行替换。例如:

```
QIcon icon = new QIcon("classpath:/images/icon.png");
if (!icon.isNull()) {
    ...
}
```

为了查找图标,Qt Jambi 将在每个目录的 images 文件夹,或者在 CLASSPATH 环境变量指定的 .jar 文件中搜寻。一旦找到 icon.png 文件,搜寻就会结束,并使用该文件。

如果没有找到该文件,也不会显示任何异常。在上面的例子中,如果没有找到 icon.png,icon.isNull()将返回 true。像 QImage 和 QPixmap 这样具有文件名参数的类,都可以由 isNull()方法来测试文件是否读取成功。如果是 QFile 的情况,可以使用 QFile.error()来查询该文件是否被读取过。

Qt Jambi 充分利用了 Java 的垃圾信息回收功能,因此不像 AWT、Swing 和 SWT,如果最后一个上层窗口的引用被删除了,那么该窗口将被回收,而不需要显式调用 dispose()。这种方法非常方便,并且工作方式也与 C++ /Qt 相同。但是对于 SDI(单文档界面)应用程序要特别小心,我们必须保持对创建的每个上层窗口的引用,以防止对它们的回收。(在 C++ /Qt 中,SDI 应用程序通常使用 Qt::WA_DeleteOnClose 属性来防止内存泄漏的产生。)

```
    public static void main(String[] args) {
        QApplication.initialize(args);
        FindDialog dialog = new FindDialog(null);
        dialog.show();
        QApplication.exec();
    }
}
```

为了方便,我们为 FindDialog 提供了 main()函数,用以实例化一个对话框,并把它弹出来。import com.trolltech.qt.gui.* 的声明可以确保使用 QApplication 对象。当 Qt Jambi 应用程序启动时,我们必须调用 QApplication.initialize(),并把命令行参数传给它。这就允许 QApplication 对象可以处理它所能识别的那些参数,例如-font 和-style。

当创建 FindDialog 时,我们把 null 作为父对象,以指明该对话框是顶层窗口。一旦 main()函数结束,对话框将失去作用,并且会被作为垃圾信息而加以回收。调用 QApplication.exec()可以开始事件循环,仅当用户关闭对话框时,控制权才会转到 main()函数中。

Qt Jambi API 与 C++ /Qt 的 API 很相似,但也有一些不同。例如,在 C++ 中,QWidget::mapTo()函数有以下特点:

```
QPoint mapTo(QWidget *widget, const QPoint &point) const;
```

QWidget 会作为非常量指针传入,而 QPoint 则会作为常量引用传入。在 Qt Jambi 中,等效的方法有以下签名:

```
public final QPoint mapTo(QWidget widget, QPoint point) { ... }
```

因为 Java 没有指针,所以在方法签名中就没有可见差异来区分传给某个方法的一个对象是否可以由该方法所修改。理论上,mapTo()函数应当可以修改任何参数,因为它们都是一些引用,但

Qt Jambi 会保证不修改 QPoint 参数，因为在 C++ 中，它是作为常量引用而传入的。从上下文中，通常清楚哪个参数可以修改，哪个不可以修改。如有疑问，可以参考文档，以澄清该情况。

除了不要修改值传递或作为引用传递的参数之外，Qt Jambi 也可以确保返回任何非 void 方法的返回值，因为在 C++ 中将返回一个值或者一个常量引用，它们是一个独立的副本，因此改变它不会产生副作用。

前面提到过，在 Qt Jambi 中，C++ /Qt 使用 QString 的地方，都可以使用 Java 的 String 来代替。对于 QChar 来说，也存在这种对应关系。在 Java 中，有两个可以与它对应：char 和 java.lang.Character。还有一些关于 Qt 容器类的类似对应关系：QHash 可以被 java.util.HashMap 代替，QList 和 QVector 可以被 java.util.List 代替，而 QMap 可以被 java.util.SortedMap 代替。此外，QThread 也可以由 java.lang.Thread 代替。

Qt 的模型/视图体系和数据库 API 广泛使用了 QVariant。该类型在 Java 中是不需要的，因为所有的 Java 对象都可以把 java.lang.Object 作为祖先，所以在 Qt Jambi 的全部 API 中，QVariant 都可以用 java.lang.Object 来代替。QVariant 提供的那些额外方法也可以使用 com.trolltech.qt.QVariant 的静态方法来代替。

这就是对 Qt Jambi 小应用程序的介绍，并且我们讨论了许多 Qt Jambi 和 C++ /Qt 程序概念上的不同点。除了 CLASSPATH 环境变量必须指定为 Qt Jambi 的安装路径之外，编译和运行 Qt Jambi 应用程序与其他 Java 应用程序并没有什么不同。我们必须用 Java 编译器编译该类，然后使用 Java 解释器运行它。例如：

```
export CLASSPATH=$CLASSPATH:$HOME/qtjambi/qtjambi.jar:$PWD
javac FindDialog.java
java FindDialog
```

这里，我们曾经使用 Bash shell 来设置了 CLASSPATH 环境变量，而其他的命令行解释器可能需要不同的语法。在 CLASSPATH 中包含当前路径就可以找到 FindDialog 类。在 Mac OS X 上，必须为 Java 指定命令行参数-XstartOnFirstThread，以使 Apple 的 Java 虚拟机可以处理线程问题。在 Windows 上，可以这样运行应用程序：

```
set CLASSPATH=%CLASSPATH%;%JAMBIPATH%\qtjambi.jar;%CD%
javac FindDialog.java
java FindDialog
```

Qt Jambi 也可以用在 IDE 中。下一节将介绍如何使用流行的 Eclipse IDE 来编辑、编译和测试 Qt Jambi 应用程序。

C.2 在 Eclipse IDE 中使用 Qt Jambi

Eclipse 集成开发环境（简称“Eclipse”）是多达 60 个开源工程的 Eclipse 家族中的一个关键软件。Eclipse 对于 Java 程序员来说很熟悉，并且由于它是用 Java 写的，因而它可以运行在所有主流平台上。Eclipse 显示的那些面板的集合称为视图（view）。Eclipse 可以使用很多视图，包括导航、大纲和编辑器视图，某一特定视图的集合则称为透视图（perspective）。

为了在 Eclipse 中可以使用 Qt Jambi 和 Qt 设计师，我们有必要安装 Qt Jambi Eclipse 集成开发包。一旦该包被释放到适当的位置，就必须运行带-clean 参数的 Eclipse 来强制它搜索新的插件，而不是依赖于它的缓存。在参数选择对话框中会出现一个“Qt Jambi Preference Page”参数。很有必要切换到该页，并且设置 Qt Jambi 的位置，这一过程可以参照 http://doc.trolltech.com/qtjambi-4.3.2_01/com/trolltech/qt/qtjambi-eclipse.html。一旦设置和验证了路径，Eclipse 将会关闭并且重新启动，以使新的设置生效。

下次启动 Eclipse 时,可以单击 File→New Project,弹出的 New Project 对话框将提供两种类型的 Qt Jambi 工程:Qt Jambi Project 和 Qt Jambi Project(Using Designer Form)。本节将通过说明 Qt Jambi 版本的 Go to Cell 例子(在第 2 章中开发的)来讨论这两种类型的工程。

要想创建一个纯代码的 Qt Jambi 应用程序,可以单击 File→New Project,选择 Qt Jambi Project,然后按向导进行操作。最后,Eclipse 将创建一个工程,生成一个 .java 框架文件,它带一个 main() 函数和一个构造函数。在 Eclipse 中,可以使用 Run→Run 菜单运行该应用程序。Eclipse 在编辑器的左边可以显示语法错误,在一个控制台窗口中可以显示运行时的那些任意错误。

创建一个使用 Qt 设计师的 Qt Jambi 应用程序与创建纯代码的应用程序类似。点击 File→New Project,然后选择 Qt Jambi Project(Using Designer Form)。将会出现一个向导。把工程命名为 JambiGoToCell,在最后一页,指定类名为 GoToCellDialog,并且选择 Dialog 作为窗体类型。

一旦向导运行结束,将生成一个 GoToCellDialog.java,以及一个 Java 用户接口文件 GoToCellDialog.jui。双击 .jui 文件会调用 Qt 设计师编辑器。将会出现一个带有 OK 和 Cancel 按钮的窗体。要访问 Qt 设计师的功能,可以单击 Window→Open Perspective→Other,然后双击 Qt Designer UI。这一透视图会显示 Qt 设计师的信号-槽编辑器、动作编辑器、属性编辑器、窗口部件箱以及其他工具箱等等,如图 C.2 所示。

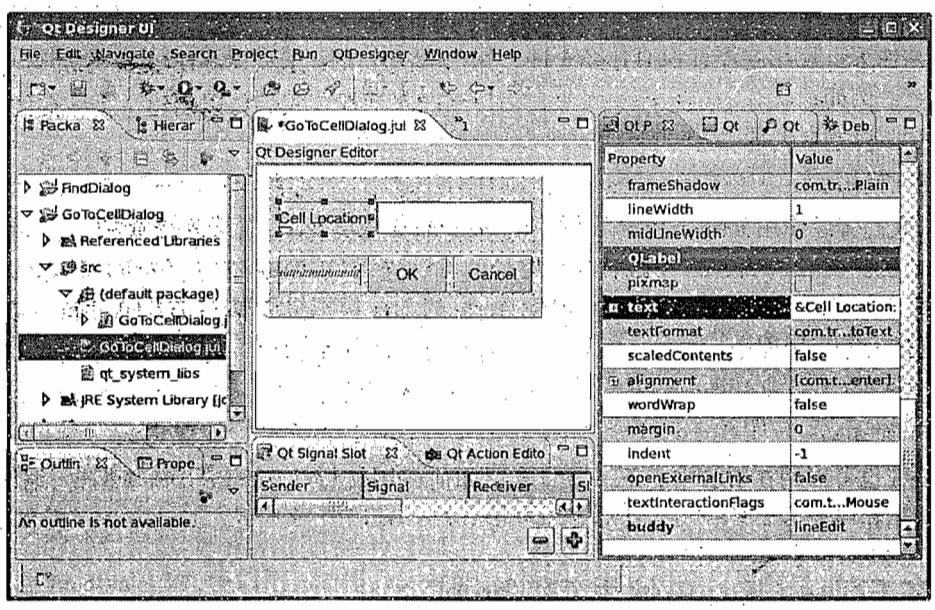


图 C.2 Eclipse 中的 Go to Cell 对话框

为了完成设计,可以执行第 2 章中同样的步骤。可以在 Qt 设计师中预览该对话框,同时由于 Eclipse 生成了框架代码,点击 Run→Run 也可以运行它。

最后的步骤是编辑 GoToCellDialog.java 文件,实现需要的功能。Eclipse 生成了两个构造函数,但只需要一个,因此删掉没有参数的构造函数。在生成的 main()方法中,必须传入 null 作为 GoToCellDialog 构造函数的父对象。同样,必须实现 GoToCellDialog(QWidget parent)构造函数,并且提供一个 on_lineEdit_textChanged(String) 方法,它可以在发送行编辑器的 textChanged()信号时得到调用。这就是 GoToCellDialog.java 文件:

```
import com.trolltech.qt.core.*;
import com.trolltech.qt.gui.*;
```

```
public class GoToCellDialog extends QDialog {
    private Ui_GoToCellDialogClass ui = new Ui_GoToCellDialogClass();

    public GoToCellDialog(QWidget parent) {
        super(parent);
        ui.setupUi(this);

        ui.okButton.setEnabled(false);

        QRegExp regExp = new QRegExp("[A-Za-z][1-9][0-9]{0,2}");
        ui.lineEdit.setValidator(new QRegExpValidator(regExp, this));

        ui.okButton.clicked.connect(this, "accept()");
        ui.cancelButton.clicked.connect(this, "reject()");
    }

    private void on_lineEdit_textChanged(String text) {
        ui.okButton.setEnabled(!text.isEmpty());
    }

    public static void main(String[] args) {
        QApplication.initialize(args);
        GoToCellDialog testGoToCellDialog = new GoToCellDialog(null);
        testGoToCellDialog.show();
        QApplication.exec();
    }
}
```

Eclipse 负责调用 juic，用来把 GoToCellDialog.jui 转换为 Ui_GoToCellDialogGlass.java，它会定义一个 Ui_GoToCellDialogClass 类，由此生成 Qt 设计师设计的对话框。我们可以创建一个该类的实例，把它的引用保存在私有成员 ui 中。

在构造函数中，我们调用 Ui_GoToCellDialogClass 的 setupUi()方法创建和布局窗口部件，设置其属性，以及信号-槽连接，包括依照名字规则 on_*objectName_signalName*() 自动连接槽。

Eclipse 集成的一个方便的特性是很容易把窗口部件箱中 QWidget 的子类拖到窗体上。我们将简单地介绍 LabeledLineEdit 自定义窗口部件，然后介绍如何在 Qt 设计师的窗口部件箱中使用它。

当使用 Qt 设计师创建自定义窗口部件时，通常要为程序员提供一些可供其修改的自定义窗口部件属性，这并不是什么难事。图 C.3 显示了一个窗体中的自定义窗口部件 LabeledLineEdit。该窗口部件有两个自定义属性，labelText 和 editText，可以在属性编辑器中设置它们。在 C++ /Qt 中，属性使用宏 Q_PROPERTY() 定义。在 Qt Jambi 中，内省机制用来检测存取函数对，它们遵循 Qt 的 *xxx*()/set*Xxx*()命名习惯，或者遵循 Java 的 get*Xxx*()/set*Xxx*()命名习惯。也可以指定其他的属性，以及使用注释自动隐藏检测到的属性。

```
import com.trolltech.qt.*;
import com.trolltech.qt.gui.*;

public class LabeledLineEdit extends QWidget {
```

第一个 import 声明是为了访问 Qt Jambi 的 @QtPropertyReader() 和 @QtPropertyWriter() 注释，它允许我们向 Qt 设计师中导入属性。

```
@QtPropertyReader(name="labelText")
public String labelText() { return label.text(); }

@QtPropertyWriter(name="labelText")
public void setLabelText(String text) { label.setText(text); }

@QtPropertyReader(name="editText")
public String editText() { return lineEdit.text(); }

@QtPropertyWriter(name="editText")
public void setEditText(String text) { lineEdit.setText(text); }
```

对于只读属性,可以简单地使用@QtPropertyReader()注释,并提供一个 getter 方法。这里我们想提供读-写属性,因此既有 getter 又有 setter。对于本例,我们忽略注释,因为访问方法依照 Qt 的命名规则。

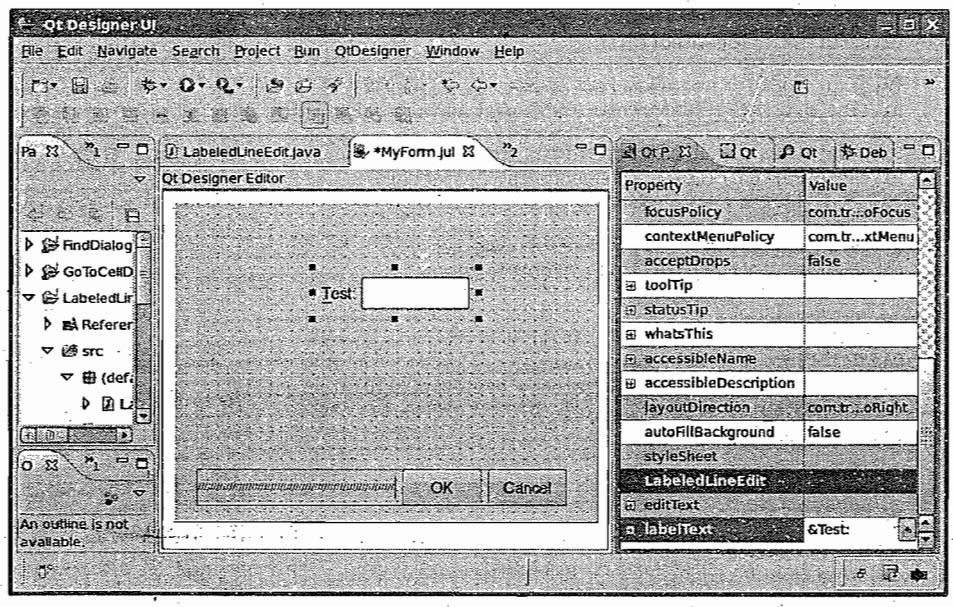


图 C.3 窗体中的 LabeledLineEdit 自定义窗口部件

```
public LabeledLineEdit(QWidget parent){
    super(parent);

    label = new QLabel();
    lineEdit = new QLineEdit();
    label.setBuddy(lineEdit);

    QHBoxLayout layout = new QHBoxLayout();
    layout.addWidget(label);
    layout.addWidget(lineEdit);
    setLayout(layout);
}

private QLabel label;
private QLineEdit lineEdit;
```

构造函数比较常规。为了能够在窗口部件箱中使用,必须提供一个带有 QWidget 参数的构造函数。

```
    public static void main(String[] args) {
        QApplication.initialize(args);
        LabeledLineEdit testLabeledLineEdit = new LabeledLineEdit(null);
        testLabeledLineEdit.setLabelText("&Test");
        testLabeledLineEdit.show();
        QApplication.exec();
    }
}
```

Eclipse 会自动生成 main()函数。我们把 null 传入 LabeledLineEdit 的构造函数,添加一行设置标签的文字属性。

一旦拥有了自定义的窗口部件,就可以将其 .java 文件复制到包浏览器中工程的 src 下。然后,调用工程的属性对话框,单击 Qt Designer Plugins 页。该页列举了工程中所有合适的 QWidget 子

类。只需选中 Enable plugin 选项就可以在窗口部件箱中看到这些子类，然后点击 OK。下次用 Qt 设计师编辑窗体的时候，就可以在窗口部件箱底部看到作为插件添加进来的窗口部件了。

这就是对于在 Qt Jambi 编程中使用 Eclipse 的简单介绍。Eclipse 提供了功能强大的便于软件开发的 IDE。正确地集成 Qt Jambi，也可以完全在 Eclipse 环境下创建 Qt Jambi 应用程序，包括 Qt 设计师的使用。Trolltech 也提供了 C++ /Qt Eclipse 集成，与 Qt Jambi Eclipse 一样具有很多优点。

C.3　在 Qt Jambi 中集成 C++ 组件

Qt Jambi 允许 C++ 程序员很容易地在 Qt 代码中集成 Java。为了使自定义的 C++ 组件可以在 Qt Jambi 中使用，可以使用 Qt Jambi 的生成器，它包含一个 C++ 头文件集以及一个 XML 文件，该文件提供了我们想要包装的类，这些类为 C++ 组件提供了 Java 绑定。Qt Jambi API 本身使用生成器生成。

图 C.4 阐述了这一过程。运行生成器之后，我们获得一些 Java 文件，必须使用 Java 编译器编译它们，还有一些 .h 和 .cpp 文件，把这些文件一起编译到 C++ 动态库中。

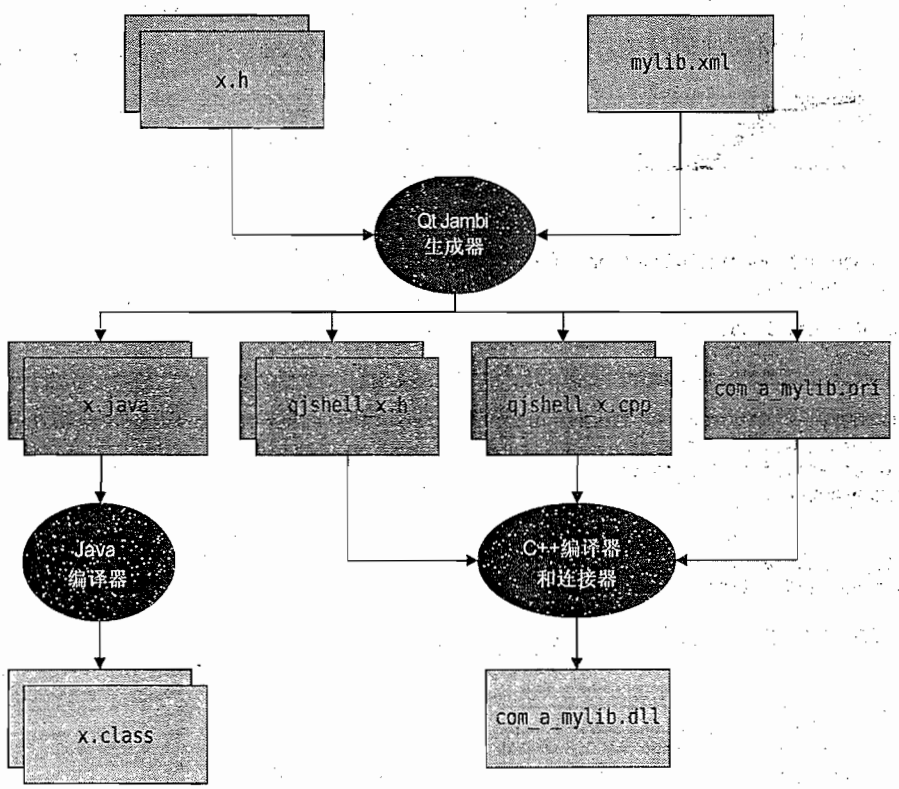


图 C.4　在 Qt Jambi 中使用 C++ 的类

为了阐述这一过程，我们将绑定第 5 章开发的 Plotter 窗口部件的 PlotSettings 类。然后，我们将在 Java 中使用绑定。图 C.5 展示了应用程序的运行情况。

头文件必须定义(或包含其定义的头文件)编译库所需的全部类：

```
#include <QtGui>
#include "../plotter/plotter.h"
```

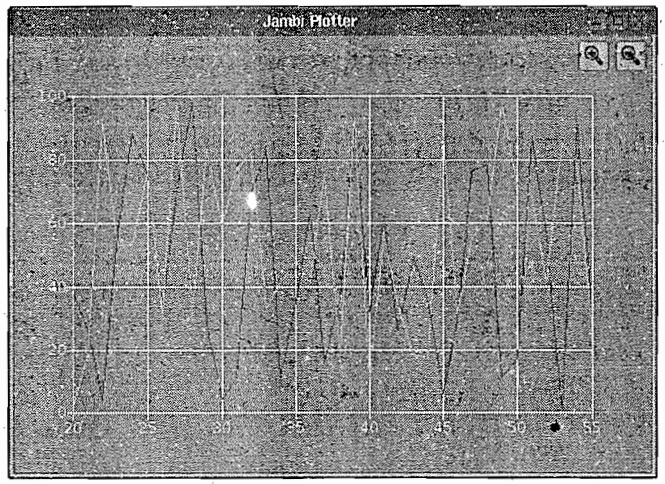


图 C.5 Jambi Plotter 应用程序

假设 Plotter 例子的代码位于 Qt Jambi 打包器的同级目录下。我们也需要一个 XML 文件,告诉生成器要打包什么,以及如何打包。文件的名称是 jambiplotter.xml:

```
<typesystem package="com.softwareinc.plotter"
            default-superclass="com.trolltech.qt.QtJambiObject">
    <load-typesystem name=":/trolltech/generator/typesystem_core.txt"
                     generate="no" />
    <load-typesystem name=":/trolltech/generator/typesystem_gui.txt"
                     generate="no" />
    <object-type name="Plotter" />
    <value-type name="PlotSettings" />
</typesystem>
```

在外层的标签中,我们为 Plotter 组件指定称作 com.softwareinc.plotter 的 Java 包。<load-typesystem>标签导入关于 QtCore 和 QtGui 模块的信息。

<object-type> 和 <value-type> 标签指定我们要使用的两个 C++ 类。我们已经指定 Plotter 类是一个 C++ "对象类型"。这适合那些不能被复制的对象,例如窗口部件。相反,PlotSettings 类被看作是 C++ 的"值类型"。

对于 Qt Jambi 用户,这两者没有明显的不同。当生成器从 C++ API 映射到 Java API 时,区别就明显了。例如,如果方法返回"值类型",那么生成器将确认返回一个新的独立的对象(为了避免副作用),但如果返回的类型是对象类型,那么将返回原始对象的引用。

我们将需要两个环境变量,一个指定 Qt Jambi 路径,另一个指定 Java 路径。在 UNIX 系统下,可以这样设置(使用 Bash shell):

```
export JAMBIPATH=$HOME/qtjambi-linux32-gpl-4.3.2_01
export JAVA=/usr/java/jdk1.6.0_02
```

在 Windows 下,应当写成:

```
set JAMBIPATH=C:\QtJambi
set JAVA="C:\Program Files\Java\jdk1.6.0_02"
```

当然,在你的系统中,版本号和目录可能不同。从现在开始,假设 JAMBIPATH 和 JAVA 都已设置好。一旦拥有头文件和 XML 文件,就可以在控制台中运行生成器:

```
$JAMBIPATH/bin/generator jambiplotter.h jambiplotter.xml
```

如果 Qt Jambi 被安装在本地目录而不是系统目录下,那么在某些系统上,命令将执行失败。

解决办法是为 Qt Jambi 的库提供一个合适的路径：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAMBIPATH/lib
```

我们又一次使用了 Bash shell 的语法。在 Mac OS X 下，相应的环境变量称为 DYLD_LIBRARY_PATH。在 Windows 下，可以将其添加到 PATH 即可：

```
set PATH=%PATH%;%JAMBIPATH%\lib
```

现在，在 Windows 下，可以像这样运行生成器：

```
%JAMBIPATH%\bin\generator jambiplotter.h jambiplotter.xml
```

代码生成器运行一段时间后，会在控制台中输出一些总结信息。在本例中，生成器会在同一目录中产生下列这些文件：

```
../com/softwareinc/plotter/PlotSettings.java
../com/softwareinc/plotter/Plotter.java
../com/softwareinc/plotter/QtJambi_LibraryInitializer.java

../cpp/com_softwareinc_plotter/com_softwareinc_plotter.pri
../cpp/com_softwareinc_plotter/metainfo.cpp
../cpp/com_softwareinc_plotter/metainfo.h
../cpp/com_softwareinc_plotter/qtjambi_libraryinitializer.cpp
../cpp/com_softwareinc_plotter/qtjambishell_PlotSettings.cpp
../cpp/com_softwareinc_plotter/qtjambishell_PlotSettings.h
../cpp/com_softwareinc_plotter/qtjambishell_Plotter.cpp
../cpp/com_softwareinc_plotter/qtjambishell_Plotter.h
```

现在，必须同时编译 Java 和 C++ 文件，但在这之前，必须确认 CLASSPATH 环境变量被正确地设置了。例如，如果使用 Bash shell，可以这样做：

```
export CLASSPATH=$CLASSPATH:$JAMBIPATH/qtjambi.jar:$PWD:$PWD/..
```

这里，使用 Qt Jambi 的 .jar 文件，并把当前目录以及当前目录的上层目录作为 CLASSPATH 的扩展。需要上层目录是为了可以访问同级的 com 目录。Windows 上的语法是：

```
set CLASSPATH=%CLASSPATH%;%JAMBIPATH%\qtjambi.jar;%CD%;%CD%\..
```

现在就可以把 .java 文件编译成 .class 文件：

```
cd ../com/softwareinc/plotter
javac *.java
```

编译完 .java 文件，必须回到 jambiplotter 目录。下一步是创建 C++ 动态库，为 Plotter 和 PlotSettings 提供 C++ 代码，还有为生成器生成的 C++ 打包代码。首先创建一个 .pro 文件：

```
TEMPLATE     = lib
TARGET       = com_softwareinc_plotter
DLLDESTDIR   = .
HEADERS      = ../plotter/plotter.h
SOURCES      = ../plotter/plotter.cpp
RESOURCES    = ../plotter/plotter.qrc
INCLUDEPATH  += ../plotter \
                $$(JAMBIPATH)/include \
                $$(JAVA)/include
unix {
    INCLUDEPATH  += $$(JAVA)/include/linux
}
win32 {
    INCLUDEPATH  += $$(JAVA)/include/win32
}
LIBS         += -L$$(JAMBIPATH)/lib -lqtjambi

include(../cpp/com_softwareinc_plotter/com_softwareinc_plotter.pri)
```

必须把 TEMPLATE 变量设置为 lib，因为我们想创建的是动态库，而不是应用程序。TARGET 变量可以指定 Java 包的名字，但使用下画线代替句点，DLLDESTDIR 指定动态库(或 DLL)放置的位

置。INCLUDEPATH 变量必须包含源码目录(因其不在当前目录下)、Qt Jambi 的包含路径、Java SDK 的包含路径和与 Java SDK 平台相关的包含路径。(附录 B 介绍了 unix 和 win32 的语法。)还需要把 Qt Jambi 的库本身包含在 LIBS 项中。最后的 include()目录用来访问生成的 C++ 文件。一旦拥有 .pro 文件,就可以运行 qmake 和 make 编译该库。

既然拥有了动态库和配套的 Java 打包代码,我们就可以在应用程序中使用它们。Jambi Plotter 应用程序创建了 PlotSettings 和 Plotter 对象,利用它们显示一些随机数据。重要的是,我们可以把它们像其他的 Java 或 Qt Jambi 类一样来使用。整个应用程序相当小,因而将会给出它的全部代码:

```
import java.lang.Math;
import java.util.ArrayList;
import com.trolltech.qt.core.*;
import com.trolltech.qt.gui.*;
import com.softwareinc.plotter.Plotter;
import com.softwareinc.plotter.PlotSettings;

public class JambiPlotter {
    public static void main(String[] args) {
        QApplication.initialize(args);

        PlotSettings settings = new PlotSettings();
        settings.setMinX(0.0);
        settings.setMaxX(100.0);
        settings.setMinY(0.0);
        settings.setMaxY(100.0);

        int numPoints = 100;
        ArrayList<QPointF> points0 = new ArrayList<QPointF>();
        ArrayList<QPointF> points1 = new ArrayList<QPointF>();
        for (int x = 0; x < numPoints; ++x) {
            points0.add(new QPointF(x, Math.random() * 100));
            points1.add(new QPointF(x, Math.random() * 100));
        }

        Plotter plotter = new Plotter();
        plotter.setWindowTitle(plotter.tr("Jambi Plotter"));
        plotter.setPlotSettings(settings);
        plotter.setCurveData(0, points0);
        plotter.setCurveData(1, points1);
        plotter.show();

        QApplication.exec();
    }
}
```

我们导入一对 Java 的标准库和 Qt Jambi 的库,以及 PlotSettings 和 Plotter 类。我们本来也可以简单地写成 import com.softwareinc.plotter.* 的形式。

初始化 QApplication 对象后,创建了一个 PlotSettings 对象,并且为它设置了一些值。C++ 版本的类,把 minX、maxX、minY 和 maxY 作为公有变量。除非特别说明,Qt Jambi 生成器会使用 Qt 的命名规则[例如,minX()和 setMinX()]为这类变量创建访问器。一旦完成绘图设置,就可以生成两组曲线数据,每组包含 100 个随机点。在 C++ /Qt 中,每条曲线的点保存在 QVector <QPointF> 中;在 Qt Jambi 中,将其保存在 ArrayList <QPointF> 中,可以依赖 Qt Jambi 对其进行转换。

设置完毕并且准备好数据后,我们就可以创建一个新的没有父对象的 Plotter 对象(使它成为上层窗口)。然后设置 plotter,导入曲线数据,调用 show()显示绘图。最后,调用 QApplication.exec()启动事件循环。

在编译应用程序时,不仅需要包含通常的 CLASSPATH,还需包含 com.softwareinc.plotter 包的路径。为了运行应用程序,必须设置几个额外的环境变量,以确保加载器可以找到 Qt Jambi 和那些绑定。在 Windows 下,需要这样设置 PATH 环境变量:

```
set PATH=%PATH%;%JAMBIPATH%\bin;%CD%
```

在 UNIX 上，必须这样设置 LD_LIBRARY_PATH：

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAMBIPATH/lib:$PWD
```

在 Mac OS X 下，环境变量称为 DYLD_LIBRARY_PATH。而且，在三个平台中，必须设置 QT_PLUGIN_PATH，由其包含 Qt Jambi 的插件目录。例如：

```
set QT_PLUGIN_PATH=%QT_PLUGIN_PATH%;%JAMBIPATH%\plugins
```

要想在所有平台上编译和运行 Jambi Plotter，有了 CLASSPATH 的设置后，只需简单输入：

```
javac JambiPlotter.java
java JambiPlotter
```

现在，我们就完成了这个例子，并且介绍了如何在 Qt Jambi 应用程序中使用 C++ 组件。事实上，Qt Jambi 生成器是 Java 使用 C++ 组件的基础，提供了许多的特性，前面的只是简单介绍。尽管生成器只支持 C++ 的一个子集，该子集包括所有最常用的 C++ 结构，包括多重继承和操作符重载。生成器的完整文档可以参阅 http://doc.trolltech.com/qtjambi-4.3.2_01/com/trolltech/qt/qtjambi-generator.html。我们将简要地介绍本例中未涉及的那些特性，以便帮你了解都有哪些内容。

影响生成器工作方式的关键是 XML 文件的内容。例如，对于 C++ 中的多重继承，在 Jambi 中可以定义一个类作为“对象类型”或“值类型”，把其他的定义成“接口类型”。

可以提供更多的方法签名。例如，除了使用 ArrayList <QPointF> 保存曲线上的点，使用 QPointF 数组会更好一些。这包括隐藏原有的 C++ 方法，使用 Java 方法替代，接收 QPointF[]，以及在后台调用原始的 C++ 方法等。可以通过修改 jambiplotter.xml 文件来做到这一点。起初，我们是利用类似于下面这样来定义 C++ 的 Plotter 类型的：

```
<object-type name="Plotter" />
```

我们将使用一个更复杂的版本来代替前面那行：

```
<object-type name="Plotter">
    <modify-function
        signature="setCurveData(int,const QVector&lt;QPointF&gt;&amp;)">
       <access modifier="private" />
       <rename to="setCurveData_private" />
    </modify-function>
    <inject-code>
        public final void setCurveData(int id,
                                com.trolltech.qt.core.QPointF points[]) {
            setCurveData_private(id, java.util.Arrays.asList(points));
        }
    </inject-code>
</object-type>
```

在 <modify-function> 元素中，我们把 C++ 的 setCurveData() 方法重命名为 setCurveData_private()，并修改其访问权限为私有，从而防止类以外方法的访问。值得注意的是，必须避免在 signature 属性中使用那些 XML 的特殊字符“<”、“>”和“&”。

在 <inject-code> 元素中，在 Java 中实现了自定义的 setCurveData() 方法。该方法接收 QPointF[] 参数，简单地调用 setCurveData_private() 方法，把数组转换为 C++ 的列表。

现在，可以用更自然的方式来创建和保存 QPointF 数组：

```
QPointF[] points0 = new QPointF[numPoints];
QPointF[] points1 = new QPointF[numPoints];
for (int x = 0; x < numPoints; ++x) {
    points0[x] = new QPointF(x, Math.random() * 100);
    points1[x] = new QPointF(x, Math.random() * 100);
}
```

余下的代码与以前的相同。

有时,在打包类时,生成器会使用 QNativePointer 产生代码。该类型是 C++ "值类型"指针的打包器,指针和数组都是可用的。文件 mjb_nativepointer_api.log 列举了那些使用 QNativePointer 生成的代码。Qt Jambi 文档建议替换所有使用了 QNativePointer 的代码,并且提供了如何替换的具体步骤。

有时需要在生成的 .java 文件中包含一些额外的 import 声明。通过使用 <extra-includes> 标签,可以很轻松地实现这一点,并且在 http://doc.trolltech.com/qtjambi-4.3.2_01/com/trolltech/qt/qtjambi-typesystem.html 中给出了完整的介绍。

Qt Jambi 生成器是一个强大的通用工具,使 Java 程序员可以使用 C++/Qt 的类,可以在单一工程中结合 C++ 和 Java 的长处。

附录 D　面向 Java 和 C# 程序员的 C++ 简介

这个附录为已经熟知 Java 或者 C# 的开发人员提供一个关于 C++ 的简短介绍。这里假定你已经熟悉了面向对象中的那些概念，如继承和多态，并且认为你也的确是想学习 C++。为了不让本书变成一部厚达 1500 页的涵盖全部 C++ 入门知识的不实用的“大部头”，所以要把这个附录仅仅限定在基本知识的范围内：只给出用来理解本书其他部分所示例子的基本知识和方法，但这些知识也足以使用 Qt 开发跨平台的 C++ 图形用户界面应用程序。

在编写这本书的时候，C++ 是开发跨平台、高性能、面向对象的图形用户界面应用程序的唯一现实选择。而一些别有用心的批评者也可能会指出，Java 或者 C# 具有更好的可用性，而 C++ 则降低了 C 的兼容性。实际上，作为 C++ 发明人的 Bjarne Stroustrup，他在“The Design and Evolution of C++”(Addison-Wesley, 1994)一书中早就指出：“即使有 C++，还可以找出更小、更简洁的语言”。

幸运的是，在使用 Qt 进行编程的时候，我们通常只关注于 C++ 的子类，这非常接近于 Stroustrup 所设想的“乌托邦”式的编程语言，从而能够让我们集中精力去解决手头的问题。此外，通过 Qt 独创性的“信号和槽”机制、对统一字符编码标准的支持以及 foreach 关键字，Qt 也在多个方面扩展了 C++。

在这个附录的第一节中，将会看到如何使用 C++ 的源文件产生一个可执行程序。这将可以引导我们探索 C++ 的一些核心概念，如编译单元、头文件、目标文件、库等，并且也可以让我们逐步熟悉 C++ 的预处理器、编译器和连接器。

然后，会转到对 C++、Java 和 C# 这些主要语言不同点的说明上：如何定义类，如何使用指针和引用，如何重载运算符，如何使用预处理器，等等。尽管 C++ 语法从表面上看与 Java 或者 C# 的语法很相似，但从深层意义上来讲，这些概念却稍微显得不尽相同。同时，作为 Java 和 C# 的创意之源，C++ 语言也与这两种语言有着诸多相同之处，包括相似的数据类型，同样的数学运算符，以及同样的基本控制流语句等。

最后一节专门用于说明标准 C++ 库，该库提供了可用于任意 C++ 程序中的完善功能。这个库是 30 多年来演化的结果，并且因此提供了涵盖程序、面向对象、函数编程风格以及宏和模板等方面的诸多方法。与 Java 和 C# 提供的库相比，标准 C++ 库的范围显得有些窄。比如，标准 C++ 库不支持图形用户界面程序设计、多线程、数据库、国际化、网络、XML 或者统一字符编码标准。要在这些领域进行开发，C++ 程序员则可能要使用各种各样(通常是与平台相关的)的库。

这就是为什么说 Qt 可以节约时间的原因。Qt 首先作为跨平台的图形用户界面工具包(一个让编写可移植图形用户界面应用程序成为可能的类的集合)而起步，但很快发展成为一个成熟的程序开发框架，它对标准 C++ 库进行了部分扩展和部分替换。尽管本书使用的是 Qt，但是如果能够知道标准 C++ 库到底提供了哪些功能也是很有用的，因为你有可能需要去处理一些使用了那些功能的代码。

D.1　C++ 入门

一个 C++ 程序由一个或者多个编译单元(compilation unit)构成。每个编译单元都是一个独立的源代码文件，通常是一个带 .cpp 扩展名(其他常用的扩展名还有 .cc 和 .cxx)的文件，编译器每

次可以处理一个这样的文件。对于每一个编译单元,编译器都会产生一个目标文件,它的扩展名是 .obj(在 Windows 中)或者 .o(在 UNIX 和 Mac OS X 中)。这个目标文件是一个二进制文件,其中包含了系统架构方面的机器代码,而程序则要运行在此基础之上。

一旦所有的 .cpp 文件都已编译完成,那么我们就可以使用一个称为连接器的特殊程序,把这些目标文件连接在一起,生成一个可执行程序。连接器会连接这些目标文件,并且会解析函数和编译单元中引用到的其他符号的内存地址。

在构建一个程序时,必须确保其中的某个编译单元包含一个 main()函数,它是程序入口的标志。这个函数不属于任何类,它是一个全局函数(global function)。图 D.1 给出了这一过程的原理图。

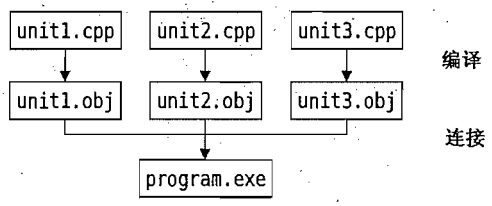


图 D.1 C++ 的编译过程(在 Windows 中)

不像 Java 的每一个源文件都必须严格包含一个类那样,C++ 可以按我们想要的形式组织各个编译单元。我们可以在同一个 .cpp 文件中实现多个类,或者也可以把一个类的实现分散到多个 .cpp 文件中,并且还可以把这些源文件命名为我们所喜欢的任意名字。当在某一个特殊的 .cpp 文件中进行修改时,只需要重新编译那个文件,然后再重新连接这个应用程序就可以生成一个新的可执行程序。

在进一步深入学习之前,让我们快速看一个 C++ 小程序的源代码,该程序可以计算一个整数的平方。这个程序由两个编译单元构成:main.cpp 和 square.cpp。

这是 square.cpp 文件中的内容:

```
1 double square(double n)
2 {
3     return n * n;
4 }
```

这个文件只简单包含了一个称为 square()的全局函数,它可以返回所带参数的平方值。

这里是 main.cpp 文件中的内容:

```
 1 #include <cstdlib>
 2 #include <iostream>

 3 double square(double);

 4 int main(int argc, char *argv[])
 5 {
 6     if (argc != 2) {
 7         std::cerr << "Usage: square <number>" << std::endl;
 8         return 1;
 9     }

10     double n = std::strtod(argv[1], 0);
11     std::cout << "The square of " << argv[1] << " is "
12               << square(n) << std::endl;
13     return 0;
14 }
```

源文件 main.cpp 包含了 main()函数的定义。在 C++ 中,这个函数的参数是一个 int 和一个 char * 数组(一个字符串数组)。可以从 argv[0]中获取程序的名字,命令行参数则分别放在 argv[1]、

argv[2]、…、argv[argc - 1]中。把参数命名为 argc(argument count,参数个数)和 argv(argument values,参数值)是一种习惯性的做法。如果这个程序不能使用命令行参数,那么可以把 main()定义成不带参数的形式。

这个 main()函数使用标准 C++ 库中的 strtod()(即"string 转换到 double")、cout(C++ 的标准输出流),和 cerr(C++ 的标准错误信息输出流),把命令行参数转换成 double,并且以文本的形式打印到终端控制台。字符串、数字和行尾标记符(endl)都是使用 << 操作符的输出流,该操作符也用于移位操作(bit-shifting)中。为了可以使用这一标准功能,我们需要在第 1 行和第 2 行中加入 #include 指示符。

标准 C++ 库中的所有函数和大多数的其他对象都在 std 命名空间中。一种访问命名空间中的某一项的方法是用命名空间的名字和::操作符作为该项名字的前缀。在 C++ 中,::操作符可以作为复杂名字的分隔符。命名空间可以使巨大的多人合作项目变得更容易些,因为命名空间可以避免命名冲突问题。在本附录的后面,还会做进一步的讨论。

位于第 3 行的代码是一个函数原型(function prototype)。它告诉编译器:存在这样一个带有给定参数和返回值的函数。而实际的函数则可以位于同一个编译单元中,也可以放在其他编译单元中。没有这个函数原型,编译器将不会让我们在第 12 行调用该函数。在函数原型中,这些参数的名字是可有可无的。

编译这个程序的过程因平台的不同而略有不同。例如,要在 Solaris 上使用 Sun C++ 编译器编译这个程序,应当输入以下命令:

```
CC -c main.cpp
CC -c square.cpp
CC main.o square.o -o square
```

最前面的两行会调用编译器生成这些 .cpp 文件对应的 .o 文件。第三行则调用连接器,并且据此生成一个称为 square 的可执行程序,于是我们就可以像下面那样来运行该程序:

```
./square 64
```

这个程序运行后会在终端上输出下列信息:

```
The square of 64 is 4096
```

要编译这个程序,你可能希望从 C++ 专家那里得到帮助。但如果无法从别人那里获得帮助,那么可以继续阅读本附录中剩余的部分而不必编译任何东西,并且可以按照第 1 章中给出的用法说明来编译你的第一个 C++ /Qt 应用程序。Qt 提供了一些工具,它们可以让你在所有平台上编译应用程序都变得轻松、简单。

再重新回到我们的程序中:在真实的应用程序中,我们通常会把 square()函数的函数原型放在一个单独的文件中,然后在需要调用这个函数的所有编译单元中都包含那个文件。这样的文件就称为头文件(header file),并且通常带一个 .h 的扩展名(常见的还有 .hh、.hpp、.hxx 等)。如果使用这种头文件的形式重写我们的程序,则需要创建一个名为 square.h 的文件,它所包含的内容如下所示:

```
1 #ifndef SQUARE_H
2 #define SQUARE_H

3 double square(double);

4 #endif
```

这个头文件被预处理命令(#ifndef、#define 和 #endif)分成三部分。这三个命令可以确保这个

头文件只作用一次,即使这个头文件在同样的编译单元中被包含了多次都是如此(当在一个头文件中又包含了其他的头文件时,就会发生这种多次包含的情况)。根据惯例,一般使用这个文件的名字作为预处理器的符号(在我们的例子中,就是SQUARE_H)。在本附录的稍后部分,还会回到预处理器这一主题上。

此时,新的 main.cpp 文件看起来像这样:

```
1  #include <cstdlib>
2  #include <iostream>

3  #include "square.h"

4  int main(int argc, char *argv[])
5  {
6      if (argc != 2) {
7          std::cerr << "Usage: square <number>" << std::endl;
8          return 1;
9      }

10     double n = std::strtod(argv[1], 0);
11     std::cout << "The square of " << argv[1] << " is "
12               << square(n) << std::endl;
13     return 0;
14 }
```

第 3 行中的 #include 命令扩展了 square.h 文件的内容。C++ 预处理器会在编译开始之前获取所有这些以“#”开始的指示符。以前,预处理器是一个单独的程序,需要在运行编译器之前先由程序员手动调用它。而现在的编译器则会隐式地调用预处理器。

第 1 行和第 2 行中的 #include 命令扩展了 cstdlib 和 iostream 头文件的内容,它们都是标准 C++ 库的一部分。标准的头文件没有 .h 后缀。包围文件名的尖括号说明这些头文件都位于系统的标准位置,而双引号则告诉编译器要到当前目录中查找头文件。这些包含命令通常都会放在 .cpp 文件内容的最前面。

不像 .cpp 文件,这些头文件自身都不是编译单元,并且也不会产生任何目标文件。头文件或许只包含一些让不同的编译单元能够互相联系的声明而已。因此,把 square() 函数的实现代码放在一个头文件中就显得有些不合适了。如果在我们的例子中那样做了,也不会产生任何不良影响,因为只包含了 square.h 一次,但是如果在多个 .cpp 文件中都包含了 square.h 文件,那么就会得到 square() 函数的多重实现(每个 .cpp 文件都包含头文件一次)。于是,连接器就会抱怨 square() 出现了多重(同样的)定义,并且会拒绝生成可执行程序。相反,如果我们声明了一个函数但是却再没有实现它,那么连接器也会报错,输出“unresolved symbol”(不可解析的符号)的错误信息。

到目前为止,我们可能会认为:一个可执行程序只是由一些目标文件构成的。但在实际情况中,可执行程序通常都会连接许多库,而这些库则可以实现许多现成的功能。库主要有两种类型:

- 静态库(static library)可以直接放进可执行程序,就好像它们也是一些目标文件一样。这可以确保不会弄丢这些库,但却会让可执行程序变得很大。
- 动态库(dynamic library,也称共享库或 DLL)位于用户机器上的标准位置,并且会在应用程序启动的时候自动加载它们。

对于以上的 square 程序,连接的是标准的 C++ 库,通常在大多数平台上都是采用这种动态库的形式实现的。Qt 自身就是一个库的集合,既可编译为静态库又可编译为动态库(默认是动态库)。

D.2　主要语言之间的差异

现在，我们将会采用一种更有条理的方式来看看 C++ 与 Java 和 C# 之间的不同之处。语言之间的这些许多不同之处都是因为 C++ 的可编译本性和对性能的追求而产生的。因此，C++ 不会在运行时检测数组是否越界，并且也没有垃圾信息收集器回收那些分配出去但是却不再使用的动态内存。

为简便起见，对于 C++ 与 Java 和 C# 中结构几乎一致的地方就不再提及了。另外，还有一些 C++ 的主题在这里也不再涉及，因为在使用 Qt 编程时并不需要它们。这其中就包括了模板类和模板函数的定义、共用体类型的定义，以及异常的使用等主题。如果要对所有这些主题都想了解，可以参考像 Bjarne Stroustrup 编著的“The C++ Programming Language”(Addison-Wesley, 2000)和 Mark Allen Weiss 编著的“C++ for Java Programmers”(Prentice Hall, 2003)等这样的一些书籍。

基本数据类型

由 C++ 语言提供的这些基本数据类型(primitive data type)与 Java 或者 C# 中的数据类型很相似。图 D.2 列出了 C++ 的基本类型以及它们在 Qt 4 所支持的平台上的定义。

C++ 类型	说　明
bool	布尔值
char	8 位整型值
short	16 位整型值
int	32 位整型值
long	32 位或者 64 位整型值
long long①	64 位整型值
float	32 位浮点值(IEEE 754)
double	64 位浮点值(IEEE 754)

图 D.2　C++ 基本类型

默认情况下，short、int、long 和 long long 数据类型都是符号型数据。也就是说，它们既可以保存负数值，又可以保存正数值。如果只需要存储非负型整数值，只需把 unsigned 关键字放在该类型前面即可。因此，一个 short 变量可以保存介于 -32 768 到 +32 767 之间的任意值，而一个 unsigned short 变量则只能保存介于 0 到 65 535 之间的任意值。如果在操作符中有这个 unsigned 操作符，那么右移操作符 >> 就会具备 unsigned(“用多个 0 填充”)的语法含义。

bool 类型可以接受的值是 true 和 false。此外，数字类型也可以用于需要 bool 值的地方，其使用规则是：0 表示 false，而其他任意的非零值表示 true。

char 类型用于存储 ASCII 字符和 8 位整型值(字节)。当用作整型值时，它可以是 signed 或者 unsigned 类型，这取决于所在的平台。类型 signed char 和 unsigned char 可用于在那些能够区分 char 正负的地方代替 char。Qt 提供了一个 QChar 类型，它可用于存储 16 位的 Unicode 字符。

① Microsoft 把非标准(因为需要标准化)的 long long 类型称为 _int64。在 Qt 程序中，可以使用 qlonglong 类型代替 long long 类型，qlonglong 则可以运行于所有 Qt 支持的平台上。

内置类型的实例不会被默认初始化。当创建一个 int 变量时,它的值应当可以明确地说是 0,但是也很有可能是 -209 486 515。幸运的是,绝大多数的编译器都会在我们试图读取未初始化变量时给予警告,并且我们也可以使用一些像 Rational PurifyPlus 和 Valgrind 这样的工具来检查运行时对未初始化内存的访问和其他与内存相关的问题。

在内存中,数值类型(long 除外)在 Qt 所支持的不同平台上都具有相同的大小,但它们的表示方法会根据系统存储字节顺序的不同而略有不同。对于高字节在后的系统架构(如 PowerPC 和 SPARC),32 位变量值 0x12345678 会存储为 4 个字节:0x12 0x34 0x56 0x78。然而,对于高字节在前的系统架构(比如 Intel x86 体系),这些字节的存储顺序就会颠倒过来。这样就会在一些需要把内存区域中的数据复制到磁盘或者是在网络上发送二进制数据的程序中产生差异。Qt 的 QDataStream 类(参见第 12 章的说明)则可用于存储与平台无关的二进制数据。

类定义

在 C++ 中,类定义(class definition)与 Java 和 C# 中的类定义相似,但是也有一些不同之处需要注意。我们将使用一系列的例子来研究这些不同点。先从一个表示(x,y)坐标对的类开始:

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
public:
    Point2D() {
        xVal = 0;
        yVal = 0;
    }
    Point2D(double x, double y) {
        xVal = x;
        yVal = y;
    }

    void setX(double x) { xVal = x; }
    void setY(double y) { yVal = y; }
    double x() const { return xVal; }
    double y() const { return yVal; }

private:
    double xVal;
    double yVal;
};

#endif
```

上面的类定义将会放在一个头文件中,通常把这个文件命名为 point2d.h。这个例子说明了 C++ 的以下几个特性:

- 类定义可以划分为 public、protected 和 private 三段,且以一个分号结束。如果没有定义段,那么就默认是 private 段。(为了保持与 C 的兼容性,C++ 提供了一个 struct 关键字,除了在没有指定段时它的默认段是 public 这一点不同外,其他都与类相同。)
- 类有两个构造函数(一个没有参数,一个则有两个参数)。我们没有声明构造函数,那么 C++ 将会自动提供一个不带参数的构造函数,并且这个构造函数的函数体为空。
- 用来获取值的函数 x()和 y()声明为 const。这就意味着它们不会(而且也不能)修改成员变量或者调用非 const 成员函数[比如 setX()和 setY()]。

上述的这些函数都实现为内联函数(inline),是类定义的一部分。还有另外一种方式是只把函

数原型放在头文件中，而把实现这些函数的代码放在 .cpp 文件中。使用这种方式时，头文件看起来应当是这样的：

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
public:
    Point2D();
    Point2D(double x, double y);

    void setX(double x);
    void setY(double y);
    double x() const;
    double y() const;

private:
    double xVal;
    double yVal;
};

#endif
```

于是就可以在 point2d.cpp 文件中实现这些函数：

```
#include "point2d.h"

Point2D::Point2D()
{
    xVal = 0.0;
    yVal = 0.0;
}

Point2D::Point2D(double x, double y)
{
    xVal = x;
    yVal = y;
}

void Point2D::setX(double x)
{
    xVal = x;
}

void Point2D::setY(double y)
{
    yVal = y;
}

double Point2D::x() const
{
    return xVal;
}

double Point2D::y() const
{
    return yVal;
}
```

文件是从包含 point2d.h 开始的，因为编译器需要在它分析类的成员函数的实现之前要先知道类的定义。然后，我们再实现这些函数，并且在函数的名字前加上以“::”操作符和类名一起构成的前缀。

我们在前面看到了如何把一个函数实现成内联函数的方法，而现在则看到了如何在 .cpp 文件中实现它。从语法上来讲，这两种方法是等效的，但是当我们调用一个声明为内联函数的函数时，绝大多数的编译器都只是简单地扩展其函数体，而不会生成实际的函数调用。这通常可以产

生更为快速的代码,但是可能会增加应用程序的大小。基于这样的原因,只有非常简短的函数才应该实现为内联函数,比较长的函数都总是应当在 .cpp 文件中加以实现。此外,如果我们忘记了某个函数的实现并试图去调用这样的函数,那么连接程序将会报错:"unresolved symbol"(不可解析的符号)。

现在尝试一下这个类:

```
#include "point2d.h"

int main()
{
    Point2D alpha;
    Point2D beta(0.666, 0.875);

    alpha.setX(beta.y());
    beta.setY(alpha.x());

    return 0;
}
```

在 C++ 中,任意类型的变量都可以直接声明而不必一定要使用 new。第一个变量会使用默认的 Point2D 构造函数(这个构造函数没有参数)进行初始化。第二个变量则使用第二个构造函数进行初始化。对一个对象的成员进行访问需要使用"."(点)操作符。

以这种方式声明变量的行为就像在 Java/C# 中声明一些基本类型一样,比如 int 和 double。例如,当使用赋值操作符时,会复制变量的内容,而不是复制对象的引用(reference)。如果要在以后修改一个变量值,那么从它那里赋值而来的其他任何变量都仍旧会保持不变。

作为一种面向对象的语言,C++ 支持继承(inheritance)和多态(polymorphism)。为了说明它们是如何工作的,我们将分析一个例子,该例以 Shape 为抽象基类,以 Circle 为子类。先从基类开始:

```
#ifndef SHAPE_H
#define SHAPE_H

#include "point2d.h"

class Shape
{
public:
    Shape(Point2D center) { myCenter = center; }

    virtual void draw() = 0;

protected:
    Point2D myCenter;
};

#endif
```

这个定义放在头文件 shape.h 中。由于在这个类的定义中引用了类 Point2D,所以需要包含头文件 point2d.h。

类 Shape 没有基类。这一点不像 Java 和 C#,C++ 没有为所有的类提供一个可以从中继承出来的一般类 Object。Qt 则为所有类型的对象提供了一个简单基类 QObject。

draw()函数的声明有两个有趣的特点:它含有 virtual 关键字,并且以"=0"为结尾。关键字 virtual 表明这个函数可能会在子类中重新得到实现。就像在 C# 中一样,C++ 的成员函数在默认情况下也是不能重新实现的。这个奇特的"= 0"的语句表明这个函数是一个纯虚函数(pure virtual function)——一个没有默认实现代码并且必须在子类中实现的函数。要把 Java 和 C# 中的"接口"的概念对应到类中,就只能用 C++ 的纯虚函数来表示了。

以下是子类 Circle 的定义:

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include "shape.h"

class Circle : public Shape
{
public:
    Circle(Point2D center, double radius = 0.5)
        : Shape(center) {
        myRadius = radius;
    }

    void draw() {
        // do something here
    }

private:
    double myRadius;
};

#endif
```

类 Circle 通过公有(public)方式继承了 Shape,也就是说,Shape 中的所有公有成员在 Circle 中仍旧是公有的。C++ 也支持保护(protected)继承和私有(private)继承,利用它们可以限制对基类的 public 成员和 protected 成员的访问。

这个构造函数带有两个参数。第二个参数是可选的,并且如果没有给定参数值就会取 0.5。构造函数在函数名和函数体之间使用一种特殊的语法把 center 参数传递给基类的构造函数。在函数体中,我们对成员变量 myRadius 进行了初始化。在基类构造函数初始化时,我们也本应在同一行初始化该变量:

```
Circle(Point2D center, double radius = 0.5)
    : Shape(center), myRadius(radius) { }
```

另一方面,C++ 不允许在类定义中初始化成员变量,因此,下面的代码就是错误的:

```
// 将无法编译通过
private:
    double myRadius = 0.5;
};
```

draw()函数与 Shape 中声明的虚函数 draw()具有相同的名字。它是该函数的一个重新实现,并且在 Circle 实例上通过 Shape 引用或者指针调用 draw()时,就会以多态的形式调用该函数。C++ 不像 C# 那样有 override 关键字。而且 C++ 也没有能够指向基类的 super 或者 base 关键字。如果需要调用一个函数的基本实现,则可以在这个函数的名字前加上一个由基类的名字和"::"操作符构成的前缀。例如:

```
class LabeledCircle : public Circle
{
public:
    void draw() {
        Circle::draw();
        drawLabel();
    }
    ...
};
```

C++ 支持多重继承,也就是说,一个类可以同时从多个类中派生出来。语法形式如下所示:

```
class DerivedClass : public BaseClass1, public BaseClass2, ...,
                     public BaseClassN
{
    ...
};
```

默认情况下,类中声明的函数和变量都与这个类的实例相关。我们也可以声明静态(static)成员函数和静态成员变量,可以在没有实例的情况下使用它们。例如:

```
#ifndef TRUCK_H
#define TRUCK_H

class Truck
{
public:
    Truck() { ++counter; }
    ~Truck() { --counter; }
    static int instanceCount() { return counter; }

private:
    static int counter;
};

#endif
```

通过这里的静态成员变量 counter,我们可以在任何时候知道还存在多少个 Truck 实例。Truck 的构造函数会增加它的值。通过前缀"~"识别的析构函数(destructor)可以减少它的值。在 C++ 中,在静态分配的变量超出作用域或者是在删除一个使用 new 分配的变量时会自动调用这个析构函数。除了我们还可以在某个特定时刻调用析构函数这一点之外,这都与 Java 中的 finalize()方法相似。

一个静态成员变量在一个类中只有单一的存在实体:这样的变量就是"类变量"(class variable)而不是"实例变量"(instance variable)。每一个静态成员变量都必须定义在 .cpp 文件(但是不能再次重复 static 关键字)中。例如:

```
#include "truck.h"

int Truck::counter = 0;
```

不这样做将会在连接时产生一个"unresolved symbol"(不可解析的符号)的错误信息。只要把类名作为前缀,就可以在该类外面访问这个 instanceCount()静态函数。例如:

```
#include <iostream>

#include "truck.h"

int main()
{
    Truck truck1;
    Truck truck2;

    std::cout << Truck::instanceCount() << " equals 2" << std::endl;

    return 0;
}
```

指针

在 C++ 中,指针(pointer)就是一个可以存储对象的内存地址的变量(而不是直接存储这个对象)。Java 和 C# 都有类似的概念——"引用"(reference),但是在语法上却并不相同。我们从研究一个精心设计的例子开始,利用它来说明指针的用法:

```
1 #include "point2d.h"

2 int main()
3 {
4     Point2D alpha;
5     Point2D beta;
```

```
 6      Point2D *ptr;

 7      ptr = &alpha;
 8      ptr->setX(1.0);
 9      ptr->setY(2.5);

10      ptr = &beta;
11      ptr->setX(4.0);
12      ptr->setY(4.5);

13      ptr = 0;

14      return 0;
15  }
```

这个例子依赖于前一小节中给出的 Point2D 类。第 4 行和第 5 行定义了两个 Point2D 对象。根据 Point2D 的默认构造函数,这两个对象将被初始化为(0,0)。

第 6 行定义了一个指向 Point2D 对象的指针。指针的语法是在变量名的前面再加上一个星号。由于没有初始化这个指针,所以它包含的是一个随机的内存地址值。通过在第 7 行给这个指针分配 alpha 对象的地址就可以解决这个初始化问题。这里的一元运算符“&”可以返回一个对象的内存地址值。地址值通常是一个 32 位或者是一个 64 位的整型值,可以用来确定一个对象在内存中的偏移量。

在第 8 行和第 9 行,我们通过 ptr 指针访问 alpha 对象。因为 ptr 是指针而不是对象,所以必须使用“ -> ”(箭头)操作符代替“.”(点)操作符。

在第 10 行,我们把 beta 的地址也赋给这个指针。于是从此时开始,通过这个指针执行的任何操作都将会影响到 beta 对象。

第 13 行把这个指针设置为空(null)指针。C++ 没有一个可以用于表示不指向对象指针的关键字。所以,我们改换用值 0(或者是符号常量 NULL,它可以扩展为 0)来代替。试图使用一个空指针会造成系统的崩溃,其提示的错误信息有“段错误”(Segmentation fault)、“常规保护错误”(General protection fault)或者是“总线错误”(Bus error)等。使用程序调试器,可以找出是哪一行代码造成了系统的崩溃。

在这个函数的最后,alpha 对象保存了坐标对(1.0, 2.5),而 beta 保存了(4.0, 4.5)。

指针通常用于存储使用 new 动态分配的对象。在 C++ 术语中,我们把这样的对象称为是分配在“堆”(heap)上,而局部变量(在一个函数中定义的变量)则存储在“栈”(stack)里。

这里给出了一段用来说明使用 new 进行动态内存分配的代码片段:

```
#include "point2d.h"

int main()
{
    Point2D *point = new Point2D;
    point->setX(1.0);
    point->setY(2.5);
    delete point;

    return 0;
}
```

new 操作符返回一个新近分配对象的内存地址。我们把这个地址存储在一个指针变量中,并且通过这指针访问该对象。当处理完这个对象后,就可以使用 delete 操作符释放它的内存。不像 Java 和 C# ,C++ 没有垃圾信息收集器,当不再需要那些动态分配的对象时,就必须明确使用 delete 来释放它们。利用第 2 章中讲述的 Qt 父-子对象机制,可以大大简化 C++ 程序中的内存管理工作。

如果忘记调用 delete,则内存就会一直保留到该程序结束时为止。这在上面的例子中不是什么大问题,因为我们只是分配了一个对象,但是如果在一个总是需要不断分配新对象的程序中,就可能造成程序总是在不断分配内存,那么就可能将机器的内存耗尽。对象一旦删除,则指向该对象的指针变量仍旧会保存这个对象的地址值。这样的指针就称为"悬摆指针"(dangling pointer),最好不要再使用这样的指针访问该对象。Qt 提供了一种"智能"(smart)指针 QPointer<T>,如果删除了它所指向的 QObject 对象,那么它就会自动把自己设置成 0。

在上面的例子中,我们调用了默认的构造函数并且调用 setX()和 setY()来初始化该对象。我们本应当使用带两个参数的构造函数来代替默认的构造函数:

```
Point2D *point = new Point2D(1.0, 2.5);
```

这个例子并不需要使用 new 和 delete。我们最好也像下面那样在栈上分配该对象:

```
Point2D point;
point.setX(1.0);
point.setY(2.5);
```

像这样分配的对象会在出现它们的程序块的末尾自动得到释放。

如果不打算通过该指针来修改这个对象,则可以把指针声明为 const 型指针。例如:

```
const Point2D *ptr = new Point2D(1.0, 2.5);
double x = ptr->x();
double y = ptr->y();

// WON'T COMPILE
ptr->setX(4.0);
*ptr = Point2D(4.0, 4.5);
```

这个常量指针 ptr 只能用于调用常量成员函数,比如 x()和 y()。当不打算使用指针修改它们时,把指针声明为 const 是一种不错的习惯。而且,如果该对象自身就是常量,那么我们就没有什么选择了,只能使用常量指针来存储它的地址值。const 的用法可以为编译器提供一定的信息,这可以提早发现一些 bug,并且也可以获得良好的性能。C# 有 const 关键字,与 C++ 的 const 关键字相似。而在 Java 中,最为接近的等价概念就是 final 了,但是它只能保护变量不被赋值,从而不能避免不在它上面调用"非常量"的成员函数。

指针既可以用在内置类型上,也可以用在类上。需要说明的是,一元运算符"*"可以返回与这个指针相关的对象的值。例如:

```
int i = 10;
int j = 20;

int *p = &i;
int *q = &j;

std::cout << *p << " equals 10" << std::endl;
std::cout << *q << " equals 20" << std::endl;

*p = 40;

std::cout << i << " equals 40" << std::endl;

p = q;
*p = 100;

std::cout << i << " equals 40" << std::endl;
std::cout << j << " equals 100" << std::endl;
```

箭头运算符"->"可用于通过指针来访问对象的成员,这纯粹是一种语法甜头(syntactic sugar)而已。除了 ptr -> member 的形式之外,我们还可以使用(*ptr).member 的形式。这里的圆括号是必需的,因为"."运算符具有比"*"运算符更高的运算优先级。

指针在 C 和 C++ 中名声不良，正是因为这一点，Java 经常借鼓吹自己没有指针而大做文章。实际上，C++ 指针在概念上与 Java 和 C# 中的引用非常相似，只是我们还可以使用指针来遍历整个内存而已——关于这一点，在这一节的后面还会讲到。此外，在 Qt 中还包含了“写时复制”(copy on write)的容器类，它具有与 C++ 一样的可在栈上实例化任意类的能力；这就意味着通常可以尽量避免指针的使用。

引用

除了指针，C++ 也支持“引用”的概念。像指针一样，一个 C++ 的引用存储的也是一个对象的地址值。两者的主要不同点在于：

- 声明引用时使用的是“&”而不是“*”。
- 引用必须是初始化过的，并且不能在后面再次重新赋值。
- 可以直接访问与引用相关联的对象，且没有像“*”或者“->”这样的特殊语法。
- 引用不能为空(null)。

在声明参数时，经常会用到引用。对于大多数类型来讲，C++ 会使用按值调用(call-by-value)的方式来作为它的默认参数传递机制。也就是说，当给一个函数传递参数的时候，该函数会接收到这个对象的一个新的副本。这里给出了一个函数的定义，它就是通过按值调用的方式来接收它的参数值的：

```
#include <cstdlib>

double manhattanDistance(Point2D a, Point2D b)
{
    return std::abs(b.x() - a.x()) + std::abs(b.y() - a.y());
}
```

于是就可以按照如下的方式来调用该函数：

```
Point2D broadway(12.5, 40.0);
Point2D harlem(77.5, 50.0);
double distance = manhattanDistance(broadway, harlem);
```

C 程序员通过把参数声明为指针而不是值的方式，能够避免不必要的复制操作：

```
double manhattanDistance(const Point2D *ap, const Point2D *bp)
{
    return std::abs(bp->x() - ap->x()) + std::abs(bp->y() - ap->y());
}
```

于是，在调用该函数的时候传递的必须是地址而不是值：

```
double distance = manhattanDistance(&broadway, &harlem);
```

C++ 引入引用的概念而使得语法变得更为简单，并且还可以使调用者避免出现传递空指针的现象。如果使用的是引用而不是指针，那么该函数看起来就会像下面这样：

```
double manhattanDistance(const Point2D &a, const Point2D &b)
{
    return std::abs(b.x() - a.x()) + std::abs(b.y() - a.y());
}
```

引用的声明与指针的声明有点相似，只是用的是“&”而不是“*”罢了。但是当我们实际使用引用的时候，无需记住它是一个内存地址，而只需把它看作是一个普通的变量就行了。另外，调用一个带有引用作参数的函数时，并不需要给予太多的考虑(不带“&”运算符)。

总而言之，通过在参数列表中把 Point2D 替换为 const Point2D &，就可以降低函数调用的开销：

不用再复制 256 位(4 个 double 值的大小),需要复制的只是 64 位或者 128 位——这取决于目标平台指针的大小。

在前一个例子中使用 const 引用,就可以让函数避免修改与这些引用相关联的对象。但是当我们需要这种特殊效果时,则可以传递一个非常量引用或者一个指针。例如:

```
void transpose(Point2D &point)
{
    double oldX = point.x();
    point.setX(point.y());
    point.setY(oldX);
}
```

在某些情况下,我们有一个引用并且需要调用一个带指针的函数,或者是相反的情形。要把引用转换成指针,可以使用一元运算符"&":

```
Point2D point;
Point2D &ref = point;
Point2D *ptr = &ref;
```

要把指针转换成引用,可以使用一元运算符"*":

```
Point2D point;
Point2D *ptr = &point;
Point2D &ref = *ptr;
```

引用和指针在内存中的表达方式一样,并且在使用时会经常互换它们,这需要根据具体问题来具体确定到底应该使用哪一种形式。一方面,引用具有更为简便的语法;另一方面,指针可以在任何时刻指向其他的任意对象,它们都可以保存一个空值,并且它们更为明晰的语法通常可以使事情变得似繁而简。基于这些原因,指针会略占上风,而用于声明函数的参数时引用则是最佳的选择,它们还可以与 const 一起配合使用。

数组

在 C++ 中,通过在变量声明和变量名字后面跟一对方括号就可以声明一个数组,但需要同时在方括号中给定数组所包含的元素个数。二维数组就是使用一维数组的数组。这里给出了一个一维数组的定义,其中包含了 10 个 int 型元素值:

```
int fibonacci[10];
```

可以采用 fibonacci[0],fibonacci[1],…,fibonacci[9]的形式对这些元素进行访问。通常情况下,我们希望以定义的方式来初始化数组:

```
int fibonacci[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

在这种情况下,可以忽略数组的大小,因为编译器能从数组的初始状态推算出元素的个数:

```
int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

静态初始化也可用于复杂数据类型,比如 Point2D:

```
Point2D triangle[] = {
    Point2D(0.0, 0.0), Point2D(1.0, 0.0), Point2D(0.5, 0.866)
};
```

如果我们无意在后续程序中修改数组,则可以让它成为常量型数组:

```
const int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

要找出一个数组中包含的元素个数,可以像下面这样使用 sizeof()运算符:

```
int n = sizeof(fibonacci) / sizeof(fibonacci[0]);
```

sizeof()运算符以字节为单位返回它的参数的大小值。一个数组中元素的个数就是它的字节数除以其中一个元素的字节数。因为这一方法非常麻烦,所以,一种更为常用的方法就是先定义一个常量,然后用这个常量来定义数组:

```
enum { NFibonacci = 10 };

const int fibonacci[NFibonacci] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

这里的本意是想把该常量声明成 const int 型变量。但遗憾的是,一些编译器已经可以用 const 变量作为数组大小的指示标记。本附录的后面会解释 enum 关键字。

数组的遍历通常是使用整数来完成的。例如:

```
for (int i = 0; i < NFibonacci; ++i)
    std::cout << fibonacci[i] << std::endl;
```

也可以使用指针来遍历数组:

```
const int *ptr = &fibonacci[0];
while (ptr != &fibonacci[10]) {
    std::cout << *ptr << std::endl;
    ++ptr;
}
```

我们用第一个元素的地址来初始化这个指针,并且一直循环到“最后一个元素之后”的元素(即“第 11 个”元素,fibonacci[10])。在每一次遍历中,“ ++ ”运算符就会把指针向后移动一次而指到下一个元素处。

不使用 &fibonacci[0],应当也可以完成对 fibonacci 的写操作。这是因为单独使用一个数组的名字会自动转换为指向该数组中的第一个元素的指针。与此相似,可以使用 fibonacci + 10 代替 &fibonacci[10]。这种方式也能够很好地工作:我们可以使用 * ptr 或者 ptr[0]获得当前元素的内容,并且可以通过 * (ptr + 1)或者 ptr[1]访问下一个元素。这一原理有时称为“指针和数组的等价性”。

为了避免出现无故的低效率,C++ 不允许我们给函数传递数组的值。相反,它们必须以地址的形式传递。例如:

```
#include <iostream>

void printIntegerTable(const int *table, int size)
{
    for (int i = 0; i < size; ++i)
        std::cout << table[i] << std::endl;
}

int main()
{
    const int fibonacci[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
    printIntegerTable(fibonacci, 10);
    return 0;
}
```

具有讽刺意味的是,尽管 C++ 没有给我们任何选择的余地,而不管我们是以地址的方式还是以值的方式来传递一个数组,但它还是在用于声明参数类型的语法中给了我们一些自由。使用的不是 const int *table,我们也本可以写作 const int table[]的方式来声明一个指针 pointer 到 constant int 的参数。与之相似的是,用于 main()中的 argv 参数则可以声明为 char *argv[]或者 char **argv。

要把一个数组复制到另一个数组,一种方法是在这个数组中进行循环:

```
const int fibonacci[NFibonacci] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
int temp[NFibonacci];
```

```
for (int i = 0; i < NFibonacci; ++i)
    temp[i] = fibonacci[i];
```

对于像 int 这样的基本数据类型,也可以使用 std::memcpy(),它可以复制一块内存中的数据。例如:

```
std::memcpy(temp, fibonacci, sizeof(fibonacci));
```

当声明一个 C++ 数组的时候,数组的大小必须是一个常数值[①]。如果希望创建一个可变大小的数组,可以有多种方法。

1. 动态分配该数组

```
int *fibonacci = new int[n];
```

这个 new []运算符会在内存的连续位置分配一定数量的元素,并且可以返回一个指向第一个元素的指针。由"指针和数组的等价性"原理可知,可以通过该指针访问元素 fibonacci[0], fibonacci[1], …, fibonacci[n-1]。当使用完数组后,应当使用运算符 delete []释放它所占用掉的内存空间:

```
delete [] fibonacci;
```

2. 使用标准的 std::vector<T> 类

```
#include <vector>

std::vector<int> fibonacci(n);
```

使用[]运算符可以访问各个元素,就像访问普通 C++ 数组一样。利用 std::vector<T>(这里的 T 是存储在向量中的元素类型值),可以在任何时候使用 resize()改变这个数组的大小,并且可以使用赋值运算符复制它。在类的名字中包含尖括号的类称为模板类。

3. 使用 Qt 的 QVector<T> 类

```
#include <QVector>

QVector<int> fibonacci(n);
```

QVector<T> 的 API 与 std::vector<T> 的 API 类似,但使用 Qt 的 foreach 关键字,它还可以支持遍历操作,并且可以使用隐式数据共享("写时复制")作为内存和速度的优化技术。第 11 章给出了 Qt 的容器类并且讲述了它们与标准 C++ 容器类之间的联系。

无论何时,或许都应当力求避免使用内置数组而应当尽量用 std::vector<T> 或者 QVector<T> 来代替它们。尽管如此,还是很有必要理解内置数组究竟是如何工作的,因为迟早或许还是希望能够把它们放进高度优化的代码中,或者还是需要利用它们与现有的 C 函数库进行交流。

字符串

C++ 中,字符串(character string)最为基本的表达方式就是使用一个以空字节('\0')为结束符的字符数组。下面的 4 个函数给出了字符串的这些工作方式:

```
void hello1()
{
    const char str[] = {
        'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\0'
    };
    std::cout << str << std::endl;
}
```

① 一些编译器允许使用上下文中的变量,但是在具备可移植性的程序中,并不鼓励使用这一特性。

```
void hello2()
{
    const char str[] = "Hello world!";
    std::cout << str << std::endl;
}

void hello3()
{
    std::cout << "Hello world!" << std::endl;
}

void hello4()
{
    const char *str = "Hello world!";
    std::cout << str << std::endl;
}
```

在第一个函数中，把字符串声明为一个数组，并且艰难地对其进行了初始化。需要注意结尾处的“\0”结束符，它表明这里是字符串的结尾。第二个函数具有相似的数组定义形式，但是这一次使用一串字符文字来初始化数组。在 C++ 中，字符串中的文字是以隐式“\0”结束符结尾的简单常量字符数组。第三个函数直接使用一串字符文字，而没有给它指定名字。但一旦转化成机器语言指令，它就与前面的两个函数一样了。

第四个函数则有一点不同，在于它创建的不仅是一个（匿名）数组而且是一个称为 str 的指针变量，并且用这个指针来存储该数组第一个元素的地址。无论这里的形式如何，从语义上来讲，这个函数与前面三个函数都一样，并且一个优化过的编译器会删除这个多余的 str 变量。

以 C++ 字符串作为参数的函数通常都带有 char *，或者 const char *。这里给出的一小段程序显示了这两种方法的用法：

```
#include <cctype>
#include <iostream>

void makeUppercase(char *str)
{
    for (int i = 0; str[i] != '\0'; ++i)
        str[i] = std::toupper(str[i]);
}

void writeLine(const char *str)
{
    std::cout << str << std::endl;
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; ++i) {
        makeUppercase(argv[i]);
        writeLine(argv[i]);
    }
    return 0;
}
```

在 C++ 中，char 类型通常保存为 8 位的值。这就是说，我们可以毫不费力地在一个 char 数组中存储 ASCII、ISO 8859-1 (Latin-1)以及其他采用 8 位编码的字符串，但是在没有使用多字节序列的情况下，就不能存储任意的 Unicode 字符。Qt 提供了强大的 QString 类，它会把 Unicode 字符串存储为 16 位 QChar 序列，同时在内部还会使用隐式数据共享（“写时复制”）的优化技术。第 11 章和第 18 章曾经详细地介绍了 QString 类。

枚举

C++ 使用颇具特色的枚举(enumeration)声明给定名字的常量,这和 C# 以及最近版本的 Java 所提供的方式相似。现在,假定我们需要在程序中存储一周中的每一天:

```
enum DayOfWeek {
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
};
```

通常情况下,我们将会把这一声明放在一个头文件中,或者甚至会放进一个类中。上面的声明可以简单地与下列常量的定义相等价:

```
const int Sunday    = 0;
const int Monday    = 1;
const int Tuesday   = 2;
const int Wednesday = 3;
const int Thursday  = 4;
const int Friday    = 5;
const int Saturday  = 6;
```

通过使用枚举结构,我们可以在后面声明一些 DayOfWeek 类型的变量或者参数,并且编译器将可以确保只对那些来自 DayOfWeek 枚举中的变量才赋值。例如:

```
DayOfWeek day = Sunday;
```

如果我们不关心类型安全性(type safety),那么也可以写作:

```
int day = Sunday;
```

需要注意的是,为了读取 DayOfWeek 枚举中的 Sunday 常量,我们只需写作 Sunday 即可,而不必写成 DayOfWeek::Sunday。

默认情况下,编译器会从 0 开始给枚举中的常数值分配连续的整型值。如果需要,也可以把它们指定成其他的常量值:

```
enum DayOfWeek {
    Sunday    = 628,
    Monday    = 616,
    Tuesday   = 735,
    Wednesday = 932,
    Thursday  = 852,
    Friday    = 607,
    Saturday  = 845
};
```

如果没有给某个枚举元素指定值,那么该元素就会用前面元素的取值加 1 来作为自己的取值。有时候,枚举用于声明整型常量,这时,我们通常会忽略枚举的名字:

```
enum {
    FirstPort = 1024,
    MaxPorts  = 32767
};
```

另外一种常见的枚举用法是用来表达选项的集合。我们以那个 Find 对话框为例,假定它有 4 个复选框,分别用来控制查询算法(使用通配符、区分大小写、向前查找以及循环查找)。我们可以用一个以 2 的幂次方为值的枚举常量来表示这一算法:

```
enum FindOption {
    NoOptions      = 0x00000000,
    WildcardSyntax = 0x00000001,
```

```
    CaseSensitive  = 0x00000002,
    SearchBackward = 0x00000004,
    WrapAround     = 0x00000008
};
```

每个选项通常都称为“标记”(flag)。我们可以使用按位“|”或者“|=”运算符来组合这些标记:

```
int options = NoOptions;
if (wilcardSyntaxCheckBox->isChecked())
    options |= WildcardSyntax;
if (caseSensitiveCheckBox->isChecked())
    options |= CaseSensitive;
if (searchBackwardCheckBox->isChecked())
    options |= SearchBackwardSyntax;
if (wrapAroundCheckBox->isChecked())
    options |= WrapAround;
```

可以使用按位“&”运算符来测试是否选中了某个标记:

```
if (options & CaseSensitive) {
    // case-sensitive search
}
```

一个类型为 FindOption 的变量每次只能包含一个标记。使用“|”对多个标记符组合的结果就是一个普通的整数。但不幸的是,这样做并不具备类型安全性:如果一个函数期望通过一个 int 参数获得 FindOptions 的组合结果,但接收的却是 Saturday,而编译器却不会对此报错。Qt 使用 QFlags <T> 来对它自己的标记类型的数据提供类型安全性保障。当我们定义一些自定义类型时,也是可以使用这个类的。更为详尽的内容请参考 QFlags <T> 的在线文档。

类型别名

C++ 允许我们使用关键字 typedef 把一个数据类型设定成其他的名字(别名)。例如,如果需要使用许多 QVector <Point2D> ,并且希望能够少敲几次键盘(或者不幸使用了我们并不熟悉的 Norwegian 键盘,这会让我们难于确定尖括号的位置),那么就可以通过把这个 typedef 声明放在某个头文件中来实现这一点:

```
typedef QVector<Point2D> PointVector;
```

从此以后,就可以把 PointVector 当做 QVector <Point2D> 的缩略形式。注意,类型的新名字需要放在旧名字的后面。typedef 的语法有些故意地模仿了变量声明的形式。

在 Qt 中,使用 typedef 主要有三个原因:

1. 方便性。Qt 可以使用 typedef 把 unsigned int 和 QList <QWidget *>声明为 uint 和 QWidgetList,从而可以大大节省击键的次数。
2. 平台差异性。在不同的平台上,某些类型需要使用不同的定义形式。例如,qlonglong 在 Windows 上定义为 _int64,而在其他平台上则定义为 long long。
3. 兼容性。Qt 3 中的类 QIconSet 在 Qt 4 中被重命名为 QIcon。为了帮助 Qt 3 的用户能够把他们的应用程序移植到 Qt 4,当启用 Qt 3 兼容性时,就会用 typedef 把 QIconSet 当作是 QIcon 的别名。

类型转换

C++ 提供了多种把变量从一种类型强制转换成另外一种类型的语法。从 C 那里继承过来的传统语法是把结果类型放在一对括号内,然后把它们再放到要转换的变量前:

```
const double Pi = 3.14159265359;
int x = (int)(Pi * 100);
std::cout << x << " equals 314" << std::endl;
```

这一语法的功能非常强大。它可用于改变指针的类型、移除 const,等等。例如:

```
short j = 0x1234;
if (*(char *)&j == 0x12)
    std::cout << "The byte order is big-endian" << std::endl;
```

在上面的例子中,我们把 short * 变量强制转换为 char * 变量,并且使用一元运算符“ * ”访问给定内存位置的字节值。在高字节在后的存储系统中,该字节的值是 0x12;在高字节在前的存储系统中,该字节的值是 0x34。由于指针和引用表达了同样的意思,所以对于上述代码我们使用引用来进行强制转换也就没有什么值得惊奇的了:

```
short j = 0x1234;
if ((char &)j == 0x12)
    std::cout << "The byte order is big-endian" << std::endl;
```

如果这个数据类型是一个类的名字、一个 typedef 或者是一个可以用单一字母数字表示的基本类型,则可以把构造函数的语法用作强制转换:

```
int x = int(Pi * 100);
```

使用传统 C 风格的强制转换方式来转换指针和引用,可以认为是一种类似于极限运动的做法,就好像在滑翔机上跳伞和利用飞机冲浪一样危险,因为编译器会让我们把任意指针(或者引用)类型都强制转换为其他的指针(或者引用)类型。正是由于这个原因,C++ 引入了 4 种具有更为准确语义的新强制转换类型。对于指针和引用,这些新风格的强制转换比那些冒险的 C 风格强制转换更为可取,并且我们早已将其用在了本书的写作中。

1. static_cast <T>() 可用于把指向 A 的指针强制转换为指向 B 的指针,其约束条件是类 B 必须是类 A 的子类。例如:

```
A *obj = new B;
B *b = static_cast<B *>(obj);
b->someFunctionDeclaredInB();
```

如果该对象不是 B 的一个实例,那么使用结果指针将可以导致莫名其妙的系统崩溃。

2. dynamic_cast <T>() 与 static_cast <T>() 类似,只是它使用的是运行时类型信息(RTTI, runtime type information)的方法来验证与这个指针相关的对象是否是类 B 的一个实例。如果不是,强制转换就会返回一个空指针 null。例如:

```
A *obj = new B;
B *b = dynamic_cast<B *>(obj);
if (b)
    b->someFunctionDeclaredInB();
```

在某些编译器中,dynamic_cast <T>() 不能跨动态库工作。它也依赖于编译器对 RTTI 的支持,但为了减小可执行文件的大小,程序员可以关闭对 RTTI 的支持这一特色。Qt 通过对 QObject 的子类提供 qobject_cast <T>() 来解决这些问题。

3. const_cast <T>() 添加或移除对指针或者引用的 const 限定。例如:

```
int MyClass::someConstFunction() const
{
    if (isDirty()) {
        MyClass *that = const_cast<MyClass *>(this);
        that->recomputeInternalData();
    }
    ...
}
```

在前面的例子中，我们舍弃了对 this 指针的 const 限定，是为了调用非 const 成员函数 recomputeInternalData()。但不推荐这种做法，并且也通常可以通过使用 mutable 关键字来避免这种做法，这一点已经在第 4 章中进行了说明。

4. reinterpret_cast <T>() 把任意类型的指针或者引用转换成任意的其他类型。例如：

```
short j = 0x1234;
if (reinterpret_cast<char &>(j) == 0x12)
    std::cout << "The byte order is big-endian" << std::endl;
```

在 Java 和 C# 中，如果需要，任何引用都可以存储成 Object 引用。C++ 则没有这种一般性的基类，但是它提供了一种特殊的数据类型：void *，它可以存储任意类型的实例的地址。在使用 void * 型数据之前，必须把它转换成另一种数据类型[可以使用 static_cast <T>()]。

C++ 提供了许多强制转换类型的方法，但是在大多数时间里，我们甚至不需任何强制转换。在使用容器类(比如 std::vector <T>或者 QVector <T>)的时候，可以指定 T 的类型并且可以解析出各个元素而不必使用强制转换。另外，对于那些基本数据类型，可以使用一些隐式转换(例如，从 char 类型转换成 int 类型)来完成数据转换，而对于自定义类型的数据，则可以通过提供单参数构造函数的方法来定义隐式转换。例如：

```
class MyInteger
{
public:
    MyInteger();
    MyInteger(int i);
    ...
};

int main()
{
    MyInteger n;
    n = 5;
    ...
}
```

对于一些单参数构造函数来说，自动转换没有多大意义。可以通过声明带 explicit 关键字的构造函数来禁用自动转换功能。

```
class MyVector
{
public:
    explicit MyVector(int size);
    ...
};
```

运算符重载

C++ 允许我们重载函数，这就意味着可以在同一作用域内用同样的名字同时声明多个函数，而只需保证它们的参数列表不同就可以了。另外，C++ 支持运算符重载(operator overloading)。也就是说，当需要在自定义的类型内使用内置运算符(比如 + 、<< 和[])时，就可以给它们分配特殊的语义。

我们已经看到了许多运算符重载的例子。当使用<<把文字输出到 cout 或者 cerr 时，我们并没有触发 C++ 的左移运算符，而是将其作为运算符的一种特殊使用形式：左侧带一个 ostream 对象(比如 cout 和 cerr)，右侧带一个字符串(或者，也可以是一个数字或者是一个流控制器，比如 endl)，返回的是该 ostream 对象，而且也允许在一行中多次调用。

运算符重载的巧妙之处在于我们可以让自定义类型的行为表现得就像使用内置类型的行为

一样。为了说明运算符重载是如何工作的,我们将会在 Point2D 对象的基础上重载 +=、-=、+ 和 - 运算符:

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
public:
    Point2D();
    Point2D(double x, double y);

    void setX(double x);
    void setY(double y);
    double x() const;
    double y() const;

    Point2D &operator+=(const Point2D &other) {
        xVal += other.xVal;
        yVal += other.yVal;
        return *this;
    }
    Point2D &operator-=(const Point2D &other) {
        xVal -= other.xVal;
        yVal -= other.yVal;
        return *this;
    }
private:
    double xVal;
    double yVal;
};

inline Point2D operator+(const Point2D &a, const Point2D &b)
{
    return Point2D(a.x() + b.x(), a.y() + b.y());
}

inline Point2D operator-(const Point2D &a, const Point2D &b)
{
    return Point2D(a.x() - b.x(), a.y() - b.y());
}

#endif
```

运算符可以被初始化为成员函数或者全局函数。在例子中,我们把 += 和 -= 实现为成员函数,把 + 和 - 实现为全局函数。

+= 和 -= 运算符带一个指向另一个 Point2D 对象的引用,并在其他对象的 x 坐标和 y 坐标基础上,对当前对象的 x 坐标和 y 坐标进行增或减运算。它们返回 * this,该值表示一个指向当前对象(它的类型是 Point2D *)的引用。利用返回的这个引用,就可以写出特殊形式的代码,比如:

```
a += b += c;
```

+ 和 - 运算符带两个参数并且通过变量(不是一个指向现存对象的引用)返回一个 Point2D 对象。inline 关键字允许我们把这些函数的定义放在头文件中。如果某个函数的函数体比较长,那么将会把该函数的函数原型放在头文件中,然后把该函数的定义(不带 inline 关键字)放在一个 .cpp 文件中。

下列代码片断给出了编程应用中 4 种运算符的重载方法:

```
Point2D alpha(12.5, 40.0);
Point2D beta(77.5, 50.0);

alpha += beta;
```

```
beta -= alpha;

Point2D gamma = alpha + beta;
Point2D delta = beta - alpha;
```

也可以像调用其他函数一样来调用运算符函数：

```
Point2D alpha(12.5, 40.0);
Point2D beta(77.5, 50.0);

alpha.operator+=(beta);
beta.operator-=(alpha);

Point2D gamma = operator+(alpha, beta);
Point2D delta = operator-(beta, alpha);
```

在 C++ 中，运算符重载是一个复杂的话题，但是在不必详知所有细节的情况下我们仍旧可以使用 C++。但了解运算符重载的基本原理还是非常重要的，因为有多个 Qt 的类（包括 QString 和 QVector <T> ）就是利用这一特性来提供一种更为简单和自然的语法的，比如字符串的连接和追加等操作。

值类型

Java 和 C# 的主要区别在于两者的值类型（value type）和引用类型（reference type）的不同。

- 值类型适用于基本类型，如 char、int 和 float，还有 C# 的 structs。区分它们的主要特征在于创建它们时并不需要使用 new。还有，在执行赋值运算时会对变量持有者进行复制。例如：

```
int i = 5;
int j = 10;
i = j;
```

- 引用类型适用于一些类，比如 Integer（在 Java 中）、String 和 MyVeryOwnClass。实例是通过 new 创建的。执行赋值运算时，只是对指向这个对象的引用的复制。要想获得深度复制（deep copy）的效果，必须调用函数 clone()（在 Java 中）或者 Clone()（在 C# 中）。例如：

```
Integer i = new Integer(5);
Integer j = new Integer(10);
i = j.clone();
```

在 C++ 中，所有类型都可以用作“引用类型”，并且那些具有可复制性的类型也还可以用作“值类型”。例如，C++ 不需要任何 Integer 类，因为我们可以像下面这样使用指针和 new：

```
int *i = new int(5);
int *j = new int(10);
*i = *j;
```

不像 Java 和 C#，C++ 会像对待内置类型一样对待用户自定义的类：

```
Point2D *i = new Point2D(5, 5);
Point2D *j = new Point2D(10, 10);
*i = *j;
```

如果想让某个 C++ 类具备可复制性，那么必须确保类有一个复制构造函数（copy constructor）和一个赋值运算符。当用同一种类型的对象初始化另外一个对象时，就会调用复制构造函数。对于这一操作，C++ 提供了两种等价的语法：

```
Point2D i(20, 20);

Point2D j(i);       // first syntax
Point2D k = i;      // second syntax
```

当在一个已经存在的变量上调用赋值运算符的时候,就会调用该赋值运算符:

```
Point2D i(5, 5);
Point2D j(10, 10);
j = i;
```

在定义一个类时,C++ 编译器会自动提供一个复制构造函数和一个赋值运算符,以用于执行成员到成员的复制。对于这个 Point2D 类,这样做就相当于在这个类的定义中写下了下列代码:

```
class Point2D
{
public:
    ...
    Point2D(const Point2D &other)
        : xVal(other.xVal), yVal(other.yVal) { }

    Point2D &operator=(const Point2D &other) {
        xVal = other.xVal;
        yVal = other.yVal;
        return *this;
    }
    ...

private:
    double xVal;
    double yVal;
};
```

对于某些类,默认的复制构造函数和赋值运算符可能都不够用。比如当这些类使用的是动态内存时,通常就会出现这种情形。要让该类具有可复制特性,就必须自己实现它的复制构造函数和赋值运算符。

对于一些不必具有可复制特性的类,可以通过让复制构造函数和赋值运算符成为私有(private)类型而禁用它们。如果随后不小心试图去复制该类的实例,那么编译器就会报错。例如:

```
class BankAccount
{
public:
    ...

private:
    BankAccount(const BankAccount &other);
    BankAccount &operator=(const BankAccount &other);
};
```

在 Qt 中,许多类都被设计用作值类(value class)。它们都有一个复制构造函数和一个赋值运算符,并且通常可以在没有 new 的堆栈上对它们进行实例化。用在这方面的例子有 QDateTime、QImage、QString 类和容器类,如 QList <T> 、QVector <T> 和 QMap <K, T> 。

还有其他的一些类可以归入"引用类型"的范畴中,特别是 QObject 以及它们的子类(如 QWidget、QTimer、QTcpSocket,等等)。这些类都有虚函数,并且也都不能被复制。例如,一个 QWidget 表示一个具体的窗口或者屏幕上的一个控件。如果内存中有 75 个 QWidget 的实例,那么屏幕上也就有 75 个窗口或者控件。这些类通常都使用 new 操作符来实例化。

全局变量和全局函数

C++ 允许声明一些不属于任何类的函数和变量,并且这些函数和变量可以被其他的任意函数访问。我们已经看到了多个全局函数的例子,包括作为程序入口的 main() 函数。全局变量还没有看到几个,因为它们需要在程序的模块和线程之间来回重复以求取折衷。但理解它们还是很重要的,因为你可能在那些 C 程序员高手和其他的 C++ 用户那里会遇到它们。

为了能够举例说明全局函数和全局变量是如何工作的，我们将会从研究一个小程序开始，该程序使用 quick-and-dirty 算法打印一个由 128 个伪随机数构成的列表。这个程序的源代码放在两个 .cpp 文件中。

第一个 .cpp 文件的源文件是 random.cpp：

```
int randomNumbers[128];

static int seed = 42;

static int nextRandomNumber()
{
    seed = 1009 + (seed * 2011);
    return seed;
}
void populateRandomArray()
{
    for (int i = 0; i < 128; ++i)
        randomNumbers[i] = nextRandomNumber();
}
```

这个文件声明了两个全局变量(randomNumbers 和 seed)和两个全局函数[nextrandomNumber()和 populateRandomArray()]。其中的两个声明包含了关键字 static，这样的声明只有在当前编译单元(random.cpp)中才是可见的，可以把这种情况称为静态连接(static linkage)。其他两个则可以从程序的任意编译单元中访问，可以把这种情况称为外部连接(external linkage)。

静态连接非常适合用于那些不需要在其他编译单元中使用的帮助函数和内部变量。它可以降低标识符(具有同样名字的全局变量或者是在不同编译单元中具有同样署名的全局函数)冲突的风险，并且可以防止那些不怀好意或者是考虑不成熟的用户访问一个编译单元的内部。

现在，让我们一起来看一看第二个文件，main.cpp，它使用了在 random.cpp 文件中用外部连接声明的两个全局变量：

```
#include <iostream>

extern int randomNumbers[128];

void populateRandomArray();

int main()
{
    populateRandomArray();
    for (int i = 0; i < 128; ++i)
        std::cout << randomNumbers[i] << std::endl;
    return 0;
}
```

在调用外部变量和函数之前，需要先声明它们。对 randomNumbers 的外部变量声明(可以让一个外部变量在当前编译单元中可见)以 extern 关键字开始。没有 extern，编译器就会认为它需要处理的是一个不确定的定义；这样就会导致连接器报错，因为同一变量同时在两个编译单元(random.cpp 和 main.cpp)中都被定义了。只要需要，可以任意多次地声明变量，但是只能定义它们一次。定义(definition)就是让编译器为该变量保留内存空间。

populateRandomArray()函数是通过使用函数原型来声明的。对于函数，extern 关键字是可有可无的。

通常情况下，会把 external 变量和函数声明放在头文件中，并且把该文件在所有需要它们的文件中包含一次：

```
#ifndef RANDOM_H
#define RANDOM_H
```

```
extern int randomNumbers[128];

void populateRandomArray();

#endif
```

我们已经看到了如何使用 static 来声明一些不属于任何一个类实例的成员变量和成员函数，并且现在也已经看到了如何使用它来声明静态连接的函数和变量。static 关键字还有另外一种用法不能忘记。在 C++ 中，可以声明一个局部静态变量(local static variable)。这样的变量会在第一次调用函数的时候得到初始化，并且会在两个函数调用时保留它们的值。例如：

```
void nextPrime()
{
    static int n = 1;

    do {
        ++n;
    } while (!isPrime(n));

    return n;
}
```

局部静态变量除了只能在定义它们的函数中可见之外，在其他方面，它们都与全局变量一样。

命名空间

命名空间(namespace)是一种用于减少 C++ 程序中名字冲突的机制。在使用了第三方软件库的大程序中，名字冲突是一个常常需要提及的话题。在你自己的程序中，可以选择是否使用命名空间。

通常，我们把命名空间放在头文件中所有声明的周围，以确保在该头文件中声明的所有标识符不会与全局命名空间中的标识符相冲突。例如：

```
#ifndef SOFTWAREINC_RANDOM_H
#define SOFTWAREINC_RANDOM_H

namespace SoftwareInc
{
    extern int randomNumbers[128];

    void populateRandomArray();
}

#endif
```

(需要注意的是，为了避免多重包含，我们对预处理器宏也进行了重命名，以便降低与不同目录中具有同样名字的头文件的冲突可能性。)

命名空间的语法与类的语法相仿，但是它不以分号结尾。这里是 random.cpp 文件的新形式：

```
#include "random.h"

int SoftwareInc::randomNumbers[128];

static int seed = 42;

static int nextRandomNumber()
{
    seed = 1009 + (seed * 2011);
    return seed;
}

void SoftwareInc::populateRandomArray()
{
    for (int i = 0; i < 128; ++i)
        randomNumbers[i] = nextRandomNumber();
}
```

与类不同,可以在任何时候“重新打开”命名空间。例如:

```
namespace Alpha
{
    void alpha1();
    void alpha2();
}

namespace Beta
{
    void beta1();
}

namespace Alpha
{
    void alpha3();
}
```

这就使定义许许多多的类成为可能,可以把这些类放在多个头文件中,然后再由这些文件构成一个命名空间。利用这一技巧,C++ 标准库就把它的所有标识符都放在 std 命名空间中。在 Qt 中,命名空间用作类似于全局形式的标识符,比如 Qt::AlignBottom 和 Qt::yellow。由于历史原因,类 Qt 不属于任何命名空间,但它使用字母“Q”作为类的开头。

要在命名空间外部引用该命名空间中的一个标识符,可以把命名空间(和一个::)作为该表示的前缀。或者,也可以使用以下三种机制之一来做到这一点,这三种方法的目的都是为了尽量减少敲击键盘的次数。

1. 定义命名空间的别名

```
namespace ElPuebloDeLaReinaDeLosAngeles
{
    void beverlyHills();
    void culverCity();
    void malibu();
    void santaMonica();
}

namespace LA = ElPuebloDeLaReinaDeLosAngeles;
```

在定义了别名之后,就可以用这个别名代替它原有的名字了。

2. 从命名空间中导入一个简单的标识符

```
int main()
{
    using ElPuebloDeLaReinaDeLosAngeles::beverlyHills;

    beverlyHills();
    ...
}
```

这里的 using 声明可以让我们从一个命名空间中访问某个给定的标识符,而不用再把该命名空间作为标识符的前缀。

3. 只用一条指令导入整个命名空间

```
int main()
{
    using namespace ElPuebloDeLaReinaDeLosAngeles;

    santaMonica();
    malibu();
    ...
}
```

使用了这种方法,似乎就更容易产生名字冲突了。如果编译器抱怨有二义性的名字(例如,在

两个不同命名空间中定义了两个具有相同名字的类),那么在需要引用这个名字的时候,通常就要使用命名空间来限定这个标识符。

预处理器

C++ 的预处理器(preprocessor)就是一个程序,它可以把 .cpp 源文件中包含的“#”指令符(比如 #include、#ifndef 以及 #endif 等)转换成不再包含那些指令符的源文件。这些指令符可以对源文件执行简单的文字处理操作,比如条件编译、文件包含以及宏扩展等。一般情况下,编译器会自动调用预处理器,但是大多数系统仍旧提供了单独调用它的方法(通常是通过编译器的-E 或者/E 选项来调用)。

1. #include 指令会把尖括号或者双引号所包含的文件扩展成它们的内容,这取决于头文件是否是安装在标准位置或者是否是当前工程中的一个部分。文件名中可能会含有“..”或者“/”(Windows 编译器可以将其正确解释为目录分隔符)。例如:

```
#include "../shared/globaldefs.h"
```

2. #define 指令定义宏。在 #define 指令之后,宏如果再次出现,那么就会在这些出现的地方用宏的定义来替换。例如,这里有一条 #define 指令:

```
#define PI 3.14159265359
```

这条指令会告诉预处理器把当前编译单元中所有出现 PI 记号的地方全部用 3.141 592 653 59 来替换。为了避免与变量和类的名字冲突,在实际应用中,宏通常命名为全部大写的字母。也有可能定义带参数的宏:

```
#define SQUARE(x) ((x) * (x))
```

在宏体内,用圆括号包围出现的所有参数是一种不错的习惯,对于整个宏体也要如此,这样就可以避免运算符优先级所出现的问题。总而言之,我们希望表达式 7 * SQUARE(2 + 3) 扩展后的结果是 7 * ((2 + 3) * (2 + 3)),而不是 7 * 2 + 3 * 2 + 3 就行了。

C++ 编译器通常也允许在命令行中使用-D 或者/D 选项来定义宏。例如:

```
CC -DPI=3.14159265359 -c main.cpp
```

在引入类型别名、枚举、常量、内联和模板功能之前,宏是一种非常流行的使用方法。现如今,宏的最主要的角色就是用于避免头文件的多重包含。

3. 可以在任何地方使用 #undef 来解除宏的定义:

```
#undef PI
```

这在我们希望重新定义宏的时候非常有用,因为预处理器不允许我们把同一个宏定义两次。在进行条件编译时,这一点也非常有用。

4. 使用 #if、#elif、#else 和 #endif 可以处理或者跳过某部分代码,这取决于宏所取的数值。例如:

```
#define NO_OPTIM          0
#define OPTIM_FOR_SPEED   1
#define OPTIM_FOR_MEMORY  2

#define OPTIMIZATION      OPTIM_FOR_MEMORY

...
#if OPTIMIZATION == OPTIM_FOR_SPEED
typedef int MyInt;
#elif OPTIMIZATION == OPTIM_FOR_MEMORY
typedef short MyInt;
#else
typedef long long MyInt;
#endif
```

在上面的例子中,只有第二个 typedef 声明才会得到编译器的处理,其结果将使 MyInt 定义为 short 的同意符。通过改变宏 OPTIMIZATION 的定义,就可以得到不同的程序。如果一个宏没有给定其定义值,那么它的值就会认为是 0。

条件编译的另外一种用法就是可以用来测试一个宏是否被定义过。像下面这样使用 defined()操作符,我们就可以做到这一点:

```
#define OPTIM_FOR_MEMORY
...
#if defined(OPTIM_FOR_SPEED)
typedef int MyInt;
#elif defined(OPTIM_FOR_MEMORY)
typedef short MyInt;
#else
typedef long long MyInt;
#endif
```

5. 为简便起见,预处理器可以识别 #ifdef X 和 #ifndef X,并且会把它们当作 #if defined(X)和 #if !defined(X)的同义词。为了避免某个头文件被多次包含,可以使用以下常用方法来包围该文件中的全部内容:

```
#ifndef MYHEADERFILE_H
#define MYHEADERFILE_H
...
#endif
```

在第一次包含该头文件时,符号 MYHEADERFILE_H 还没被定义过,因而编译器就会处理 #ifndef 和 #endif 之间的所有代码。而在第二次或者是以后再次包含该头文件时,由于已经定义了符号 MYHEADERFILE_H,所以就会跳过整个 #ifndef … #endif 代码段。

6. #error 指令可以在编译时给出用户自定义的错误信息。这通常与条件编译一起用来报告可能出现错误的地方。例如:

```
class UniChar
{
public:
#if BYTE_ORDER == BIG_ENDIAN
    uchar row;
    uchar cell;
#elif BYTE_ORDER == LITTLE_ENDIAN
    uchar cell;
    uchar row;
#else
#error "BYTE_ORDER must be BIG_ENDIAN or LITTLE_ENDIAN"
#endif
};
```

不像其他大多数的 C++ 结构体,其中的空格不起任何作用,预处理器指令需要独占一行,且不需要用分号结尾。对于非常长的指令,可以通过在除最后一行之外的每一行的末尾加上一个反斜杠方法把指令分隔成跨行的表示形式。

D.3　C++ 标准库

这一节将简单讨论 C++ 标准库。图 D.3 列出了 C++ 中的核心头文件。头文件 <exception>、<limits>、<new> 和 <typeinfo> 支持 C++ 语言,例如,<limits> 允许我们测试编译器的整数和浮点数在数学方面的支持情况,<typeinfo> 则为我们提供了基本的内省(introspection)功能。其他的头文件

也通常都提供了一些有用的类,包括一个字符串类和一组复杂的数值类型。由 <bitset>、<locale>、<string> 和 <typeinfo> 提供的功能大致可以与 Qt 中的 QBitArray、QLocale、QString 和 QMetaObject 类相当。

头文件	说 明
<bitset>	用于表示固定长度位序列的模板类
<complex>	用于表示复杂数值的模板类
<exception>	与异常处理相关的类型和函数
<limits>	用于指定值类型属性的模板类
<locale>	与本地化相关的类和函数
<new>	用于管理动态内存分配的函数
<stdexcept>	预定义的用于报告错误的异常类型
<string>	字符串容器模板和以字符处理为主的模板
<typeinfo>	用于提供类型的基本元信息的类
<valarray>	用于表示数值数组的模板类

图 D.3 C++ 核心库中的头文件

标准 C++ 也包含了一组用于处理输入/输出的头文件,列在图 D.4 中的表格里。这些标准输入/输出类设计于 20 世纪 80 年代并且也不算复杂,这使得非常难于扩展它们——实际上,仅就只说明这个困难性就足以写上几本书。同时,它还给程序员留下了一个潘多拉魔盒,其中包含了许多与字符编码相关和基于平台的基本数据类型的二进制数据表示方法等许多问题。

头文件	说 明
<fstream>	用于处理外部文件的模板类
<iomanip>	带一个参数的输入/输出流操作符
<ios>	用于输入/输出流的模板基类
<isofwd>	用于多个输入/输出流模板类的前置声明
<isotream>	标准输入/输出流(cin、cout、cerr、clog)
<istream>	控制从流缓存中输入的模板类
<ostream>	控制输出到流缓存中的模板类
<sstream>	与字符串流缓存相关的模板类
<streambuf>	缓存输入/输出操作的模板类
<strstream>	对字符数组执行输入/输出流操作的类

图 D.4 与 C++ 输入/输出库相关的头文件

第 12 章给出了相应的 Qt 类,其中给出了富有特色的统一字符编码标准输入/输出和传统大字符集的字符编码,还给出了与平台无关的用于存储二进制数据的抽象类。Qt 的输入/输出类构成了 Qt 处理进程之间的通信、网络和 XML 支持的基础。Qt 的二进制和文本流类都非常容易扩展成处理自定义数据类型的类。

20 世纪 90 年代初期出现了标准模板库(STL, Standard Template Library)的介绍,它是一个基于模板的容器类、迭代器类和在最后一刻才归入 ISO 标准 C++ 的算法的集合。图 D.5 给出了构成标准模板库的头文件列表。标准模板库非常简约,几乎全是提供一般类型安全功能的数学设计。Qt 提供了它自己的容器类,这些类的部分设计灵感就来自于标准模板库。至于这些,已经在第 11 章中进行了阐述。

头文件	说　明
<algorithm>	一般用途的模板函数
<deque>	双端队列模板容器
<functional>	用于帮助构造和操作运算符的模板
<iterator>	用于帮助构造和操作迭代器的模板
<list>	双向链表模板容器
<map>	单值和多值映射模板容器
<memory>	用于简化内存管理的工具
<numeric>	数字操作模板
<queue>	队列模板容器
<set>	单值和多值集模板容器
<stack>	栈模板容器
<utility>	基本模板函数
<vector>	向量模板容器

图 D.5　STL 的头文件

由于 C++ 从本质上来说是 C 程序设计语言的一个超集，C++ 程序员也可以随意调用整个 C 库。既可以通过传统方式(例如，<stdio.h>)使用 C 头文件，也可以使用带一个 c 作为前缀并且没有 .h 后缀的新风格(例如，<cstdio>)使用 C 头文件。当使用新风格方式时，这些函数和数据类型都是在 std 命名空间中声明的。(这不能用于像 ASSERT()这样的宏，因为预处理器不会指导命名空间的存在。)如果你的编译器支持新风格的表达方式，那么就建议你使用这种方式。

图 D.6 列出了用于 C 库的头文件。它们中的大多数都提供了与最新版本的 C++ 或者 Qt 的头文件相同的功能。但有一个例外值得注意，它就是 <cmath> ，只有它声明了一些诸如 sin()、sqrt() 和 pow()这样的数学函数。

头文件	说　明
<cassert>	ASSERT()宏
<cctype>	用于分类和映射字符的函数
<cerrno>	与报告错误条件相关的宏
<cfloat>	用于指定基本浮点类型属性的宏
<ciso646>	ISO 646 用户字符集的备份拼写法
<climists>	用于指定基本整型类型属性的宏
<clocale>	与本地化相关的函数和类型
<cmath>	与数学相关的函数和常量
<csetjmp>	用于执行非局部跳转的函数
<csignal>	用于处理系统信号的函数
<cstdarg>	用于实现参数列表函数的宏
<cstddef>	用于多个标准头文件的一般定义
<cstdio>	用于执行输入/输出的函数
<cstdlib>	一般实用功能方面的函数
<cstring>	用于处理字符数组的函数
<ctime>	用于处理时间的类型和函数
<cwchar>	扩展的多字节和大字符实用程序
<cwctype>	用于分类和映射大字符的函数

图 D.6　C++ 为 C 库提供的一些头文件

至此,就完成了对 C++ 标准库的简要回顾。在因特网上,Dinkumware 为 C++ 标准库提供了完整的参考文档,网址是:http://www.dinkumware.com/refxcpp.html,并且 SGI 在 http://www.sgi.com/tech/stl/为程序员提供了全面的 STL 编程指南。C++ 标准库的官方定义可以在"the C and C++ standards"中找到,可以从国际标准化组织(ISO,International Organization for Standardization)获得该文档的 PDF 格式的文件或者纸质副本。

在这一附录中,我们已经快速涉猎了许多有关 C++ 的基础性东西。当从第 1 章开始学习 Qt 的时候,你应当发现其中使用的语法还是要比这个附录中所提到的语法简单和有条理得多。在良好的 Qt 程序设计中,只需要使用 C++ 的子类,并且通常可以避免那些更为复杂的需求和那些可能让 C++ 变得晦涩难懂的语法。一旦在你开始键入代码并且编译和运行可执行程序的时候,清晰而简便的 Qt 方法就会展现出来。并且,一旦开始编写更具挑战性的程序,尤其是在编写那些需要兼具速度和技巧的图形程序时,C++/Qt 的组合就能够轻而易举地满足你的需求。

反侵权盗版声明

电子工业出版社依法对本作品享有专有出版权。任何未经权利人书面许可，复制、销售或通过信息网络传播本作品的行为；歪曲、篡改、剽窃本作品的行为，均违反《中华人民共和国著作权法》，其行为人应承担相应的民事责任和行政责任，构成犯罪的，将被依法追究刑事责任。

为了维护市场秩序，保护权利人的合法权益，我社将依法查处和打击侵权盗版的单位和个人。欢迎社会各界人士积极举报侵权盗版行为，本社将奖励举报有功人员，并保证举报人的信息不被泄露。

举报电话：（010）88254396；（010）88258888

传　　真：（010）88254397

E-mail　：dbqq@phei.com.cn

通信地址：北京市万寿路 173 信箱

　　电子工业出版社总编办公室

邮　　编：100036